Operational Semantics for Secure Interoperation

Adriaan Larmuseau
Uppsala University, Sweden
first.last@it.uu.se

Marco Patrignani
iMinds-Distrinet, K.U. Leuven, Belgium
first.last@cs.kuleuven.be

Dave Clarke
Uppsala University, Sweden
iMinds-Distrinet, K.U. Leuven, Belgium
first.last@it.uu.se

Abstract

Modern software systems are commonly programmed in multiple languages. Research into the security and correctness of such multi-language programs has generally relied on static methods that check both the individual components as well as the interoperation between them. In practice, however, components are sometimes linked in at run-time through malicious means. In this paper we introduce a technique to specify operational semantics that securely combine an abstraction-rich language with a model of an arbitrary attacker, without relying on any static checks. The resulting operational semantics, instead, lifts a proven memory isolation mechanism into the resulting multi-language system. We establish the security benefits of our technique by proving that the obtained multi-language system preserves and reflects the equivalences of the abstraction-rich language. To that end a notion of bisimilarity for this new type of multi-language system is developed.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Interoperability, Security

Keywords memory protection, multi-language semantics, bisimulation, fully abstract compilation

1. Introduction

Modern software systems consist of numerous interoperating components written in different source languages. Reasoning about the semantical properties of such a system is often done by developing a combined language composed of the models of the source languages.

Securing a multi-language software system requires, at least, that the combined language preserves the abstractions of each of the composed languages. This is because language based security relies on a notion of equivalence: no client of a component should be able to distinguish between two different implementations of that component if all manifestations of the differences between the implementations are masked by language abstractions. A client of two λ-calculus terms $(\lambda x.(iszero\ x)\ \&\ true)$ and $(\lambda x.(iszero\ x))$, for example, cannot distinguish between the two as the λ-term abstracts away the implementation details of its subterm.

When the abstractions of the composed languages are not preserved, a malicious interoperating component can thus violate the security of the other components [1].

Previous approaches to securing multi-language software systems have relied on static methods that check both the components individually as well as the interoperation between them [8, 9, 23]. However because some software components may be dynamically linked at run-time, written in languages with no abstractions or susceptible to code injection attacks, these static solutions are easily circumvented in practice [17]. Current multi-language security techniques thus do not preserve the abstractions of the composed languages when faced with a component that is malicious. We refer to such a malicious component as the arbitrary machine-level attacker.

To study this problem, this paper introduces a technique for specifying operational semantics that enable an abstraction-rich language to interoperate securely with a model of an arbitrary machine-level attacker without relying on any static checks on the components not written in the abstraction-rich language. To that end this paper lifts a proven memory isolation mechanism that protects a certain memory area by restricting access to that area through a set of designated entry points, into the semantics of the combined language (Section 2). Efforts are underway to embed such a memory isolation mechanism into future commercial Intel processors [16].

We illustrate our technique by securing the interoperation between the simply typed λ-calculus, hereafter referred to as the λ_s-calculus and a model for an arbitrary machine-

PLAS '14, July 29, 2014, Uppsala, Sweden.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

level attacker. This machine-level attacker is modeled as a λ-calculus extended with the syntactical equality operator $\equiv$, referred to as the λ_a-calculus. Given how precise such an operator is at distinguishing components, we argue that the λ_a-calculus is a good model of an arbitrary attacker.

We provide an informal overview to our method by first combining the λ_a- and λ_s-calculus into the combined language λ_m, using the commonly used multi-language semantics of Matthews and Findler [15] (Section 3). This combined language, however, does not preserve the abstractions of the λ_s-calculus. Lifting the memory isolation mechanism into the λ_m-calculus to resolve this security issue fails due to the direct syntactical embedding that Matthews and Findler's method relies on.

This issue is resolved by removing the direct syntactical embedding between the λ_s- and λ_a-calculus (Section 4). Instead the interoperation between both calculi is encoded into partial evaluation stacks similar to how continuation passing style conversion makes continuations explicit. The resulting combined language is referred to as the λ^+-calculus.

To establish that the resulting combined language λ^+ is capable of preserving the abstractions of the λ_s-calculus, a secure/fully abstract compilation scheme from the λ_s-calculus to the λ^+-calculus is introduced. This compilation scheme preserves and reflects the equivalences of the λ_s calculus (Section 5). Contextually equivalent terms in the λ_s-calculus are thus compiled to contextually equivalent terms in the λ^+-calculus and all compiled terms that are contextually equivalent in the λ^+-calculus are contextually equivalent in the λ_s-calculus.

Because directly proving contextual equivalence is complex [22], we develop bisimulations that coincide with contextual equivalence for the λ_s- and λ^+-calculus. The fact that this compilation scheme is indeed fully abstract is established by systematically relating the states of the bisimulations over the λ_s- and λ^+-calculi.

This paper makes the following contributions:

- Operational semantics that ensure secure interoperability between the simply typed λ-calculus and an arbitrary machine-level attacker.
- A bisimulation over the produced combined language.
- A fully abstract compilation scheme from the simply typed λ-calculus to the combined language that results from our technique.

This technique for specifying operational semantics for secure interoperation is currently limited to multi-language software systems between two languages as well as languages that are not concurrent. In the future, however, this technique could be extended to multi-language software systems that securely interoperate between a number of complex and concurrent languages.

2. A Memory Isolation Mechanism

Preserving the abstractions of a source language from an arbitrary machine-level attacker has been achieved by employing a memory isolation mechanism [4, 19]. In this paper we lift a low-level isolation mechanism referred to as Protected Modules Architecture (PMA) into the operational semantics of a multi-language system.

PMA is a fine-grained, program counter-based, memory access control mechanism that divides memory into protected and unprotected memory. The protected memory is further split into two sections: a protected code section accessible only through designated entry points, and a protected data section that can only be accessed by the code section. As such the unprotected memory is limited to executing the code at the entry points, neither the code nor the data of the protected memory can be executed, written or read by the unprotected memory. An overview of the access control mechanism between the protected and unprotected memory is given in Table 1.

From \To	**Protected**			**Unprotected**
	Entry	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Table 1: PMA protects its data by forcing the unprotected memory to use the designated entry points.

Note that this technique could be considered to be the dual of sandboxing. When securing a system through sandboxing it is the attacker that is placed within a confined memory area, while the secure program operates as usual.

A variety of PMA implementations exist. While most of them are research prototypes [18, 21], Intel is developing a new set CPU instructions, referred to as SGX, that enable the creation of protected modules in commercial processors [16].

3. Informal Overview

A combined language must preserve and reflect the equivalences of the combined languages. This section firstly details why the λ_a-calculus is an accurate model of an arbitrary machine-level attacker (Section 3.1). The λ_a-calculus is then combined with the simply typed λ-calculus (λ_s) by applying Matthews and Findler's natural embedding, resulting in a calculus that we refer to as the λ_m-calculus (Section 3.2). This λ_m-calculus is, however, not capable of preserving the equivalences of the λ_s-calculus. To resolve that issue we attempt to lift the PMA mechanism into the calculus but fail (Section 3.3). The failures encountered point out how to develop a combined language that can lift the PMA mechanism (Section 3.4).

The terms, types and contexts of the λ_s-calculus are typeset in a **bold black font**. The terms and contexts of the λ_a-calculus, are typeset in a grey-sans serifs font .

3.1 The λ_a-Calculus as an Attacker Model

As mentioned previously, language based security relies solely on a notion of equivalence. The λ_a-calculus, an untyped lambda calculus with a syntactical equality operator, is a relevant attacker model as it has no equivalences other than trivial syntactical equalities. For any λ_a-term t there exists a λ_a-context C that can distinguish t from any term that differs from it syntactically as follows.

$$\mathsf{C} = (\mathsf{t} \equiv [\cdot])$$

where $\equiv$ is a syntactical equality operator.

While the syntactical equality operation may seem like an overly strong attacker model, we argue that the syntactical equality operator simply reflects a machine-level attackers ability to distinguish between any combination of bits that it has access to. We do not extend the attacker with the ability to modify or introduce terms dynamically. This is not necessary as previous work by Wand [25] has established that inspection alone is sufficient as an attacker model, extending the attacker does not strengthen it.

3.2 A Natural Embedding of the λ_s- and λ_a-calculus

The natural embedding introduces two new terms into the combined λ_m-calculus: $\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{t}$ and ${}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\ \mathsf{t}$. The former is a λ_a-term that embeds a λ_s-term $\mathbf{t}$ and the latter is a λ_s-term that embeds a λ_a-term t. Both terms are annotated with a λ_s-type $\boldsymbol{\sigma}$. These type annotations are used to perform dynamic typechecks on the interaction between the terms of the λ_a- and λ_s-calculi to ensure that the typing properties of the λ_s-calculus are preserved.

In the λ_m-calculus primitive values are simply converted into the respective representation when they transition between the composed languages. Function calls rely on a wrapping mechanism: when the λ_s-calculus, for example, gains access to a λ-term of the λ_a-calculus it wraps that λ-term into a new λ_s-calculus λ-term as follows:

$$^{\boldsymbol{\sigma}\rightarrow\boldsymbol{\sigma}'}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\mathsf{t}) \rightarrow \lambda \mathbf{y} : \boldsymbol{\sigma}.(^{\boldsymbol{\sigma}'}\mathbf{S}\mathsf{A}\ ((\lambda \mathsf{x}.\mathsf{t})\ (\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{y})))$$

The λ_m-calculus is not capable of preserving the abstractions of the λ_s-calculus. Take for example the following two λ_s-calculus terms.

$$\mathbf{t_{IF}} = (\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{if}\ \#\mathbf{t}\ \mathbf{then}\ \mathbf{x}\ \mathbf{else}\ \mathbf{x})$$
$$\mathbf{t_{ID}} = (\lambda \mathbf{y} : \boldsymbol{\sigma}.\mathbf{y})$$

The terms $\mathbf{t_{IF}}$ and $\mathbf{t_{ID}}$ are contextually equivalent in the λ_s-calculus as there is no λ_s-calculus context that can distinguish them. However in the combined λ_m-calculus these two terms are no longer contextually equivalent as the following λ_m-calculus context can distinguish between them.

$$\mathsf{C_{ID}} = (\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}\rightarrow\boldsymbol{\sigma}}\ (\lambda \mathbf{y} : \boldsymbol{\sigma}.\mathbf{y}) \equiv \mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}\rightarrow\boldsymbol{\sigma}}\ [\cdot])$$

The problem at hand is that $\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{t}$ is a λ_a-term whose contents can be compared against any other λ_a-term, in the same way that a low-level attacker can compare any two sets of bits that it has access to.

3.3 Lifting PMA into the λ_m-Calculus

This paper aims to preserve the abstractions of the source languages of a multi-language software system by lifting the memory isolation model of PMA into the combined language. To that end the λ_m-calculus is investigated as a model of PMA. Note that we are aware of the fact that operational semantics of λ_m-calculus were explicitly designed to abstract away low-level details such as memory models. The goal of this section is to clarify our reasons for introducing a new operational semantics.

Clearly the λ_m-calculus is capable of modeling the split memory model of the PMA mechanism: simply assume that the terms of the λ_s-calculus reside in the protected memory and that the terms of the λ_a-calculus reside in the unprotected memory. By extension the λ_a-term $\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{t}$ represents an entry point to a λ_s-calculus term and the λ_s-term ${}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\ \mathsf{t}$ represents an unprotected call to a λ_a-calculus term.

This model, however, is not precise enough. A first issue is the use of $\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{t}$ to model the entry point mechanism of PMA. Reconsider our previously problematic λ_m-context $\mathsf{C_{ID}}$. Whether or not this context is able to distinguish between $\mathbf{t_{IF}}$ and $\mathbf{t_{ID}}$ when $\mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{t}$ is assumed to be entry point to a protected piece of memory relies, in practice, on what the binary values of the entry points are.

The λ_m-calculus, however, does not allow us to reason about the binary values of the entry points. This limitation also artificially restricts the attacker model: in the λ_m-calculus the attacker can only manipulate the entry points that have been shared with it. In practice, however, an attacker is capable of calling any existing entry point by guessing its address.

A second issue is the way the λ_m-calculus wraps the insecure functions it gains access to. As illustrated in Figure 1, a memory representation of what happens when a λ_s-calculus program is given a λ_a-calculus function $(\lambda \mathsf{x}.\mathsf{t})$, every time a λ_s-program is given a λ_a-function it wraps that function into

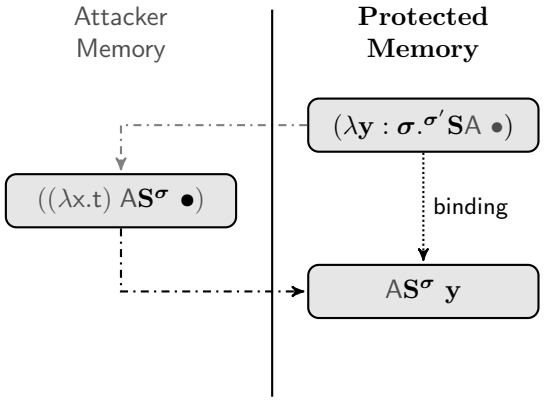

Figure 1: The sub-terms of: $\lambda \mathbf{y} : \boldsymbol{\sigma}.^{\boldsymbol{\sigma}'}\mathbf{S}\mathsf{A}\ ((\lambda \mathsf{x}.\mathsf{t})\ \mathsf{A}\mathbf{S}^{\boldsymbol{\sigma}}\ \mathbf{y})$ are spread across protected and unprotected memory.

a lambda function of its own and as a result writes out a new chunk of memory to the unprotected memory space. As required by λ_m this memory chunk encodes an application of the received function to an entry point to the bound variable of the enclosing λ-term. A λ_s-calculus program thus writes out a chunk of memory to the unprotected memory when it receives an insecure function, irrespective of whether or not it immediately calls that function.

In practice, an attacker in the unprotected memory will be able to both observe and execute that memory chunk before the shared function is used. The λ_m-calculus is not capable of modeling such an attack, thus leaving the consequences of the attack open to the implementation. We argue that this way of modelling function calls raises more questions and possible security problems than it resolves.

3.4 Our Approach to Modeling the PMA Mechanism

We propose to resolve the issues that plague the λ_m-calculus by removing the direct syntactical embedding (Section 3.4.1), modeling the entry points as a naming mechanism (Section 3.4.2) and simplifying function calls (Section 3.4.3).

3.4.1 Removing Syntactical Embedding

As in Section 3.3, assume that the terms of the λ_s-calculus reside in the protected memory of the PMA mechanism and that the terms of the λ_a-calculus reside in unprotected memory. We represent this assumption literally by grouping λ_a-terms and λ_s-terms at their respective side of a fixed syntactical boundary as follows:

$$\lambda_{\mathsf{a}}\text{-terms} \;||\; \lambda_{\mathbf{s}}\textbf{-terms}$$

Because we limit our technique to sequential languages, only one term on one side of the program can be executing at any given moment. In order to enforce this, all terms outside of the term that is executing must feature a hole: $[\cdot]$.

These holes encode the call stack between the λ_a-calculus and the λ_s-calculus, as done previously in Jeffrey and Rathke's fully abstract trace semantics for Java Jr [14]. Each hole in a λ_a-term is thus only to be filled by a term from the λ_s-calculus and vice versa.

3.4.2 Entry Points as Names

To more accurately model the entry points mechanism we extend the λ_a-calculus with enumerable names $\mathsf{n_i}$ that denote the λ_s-calculus terms that are accessible to terms of the λ_a-calculus. Programs in the λ_a-calculus can compare and construct these names, thus removing the previous limitations of the λ_m-calculus.

Reconsider, once again, our previously problematic λ_m-context. It is now defined as follows:

$$\mathsf{C_{ID}} = (\mathsf{n_r} \;\equiv [\cdot]) \;||\; \lambda_{\mathbf{s}}\textbf{-terms}$$

where the name $\mathsf{n_r}$ is a random guess by the attacker. The question now is whether or not the context $\mathsf{C_{ID}}$ could ever guess a name that allows it to distinguish between the λ_s-terms $\mathbf{t_{IF}}$ and $\mathbf{t_{ID}}$.

Our approach to protecting from this attack is to deterministically create a new name every time a λ_s-term is shared with the λ_a-calculus, effectively enumerating the shared functionality. Two λ_s-programs will thus share the same set of names with a λ_a-calculus context if and only if they share the same number of λ-terms with that context.

Our example context $\mathsf{C_{ID}}$ is thus unable to distinguish between $\mathbf{t_{IF}}$ and $\mathbf{t_{ID}}$ or any other two λ_s-terms that share only one name. Even though it can easily guess the deterministically created names.

Note that this does not mean that any two λ_s-calculus terms that share the same number of λ-terms to a λ_a-context, will be indistinguishable to that context. The values and function calls that two λ_s-calculus terms share with a λ_a-calculus context can still be observed and distinguished.

Note again that these names are terms of the λ_a-calculus, not of the λ_s-calculus. As such they do not prohibit a full abstraction result, as shown in Section 5.3.

Also note that we restrict the usage of names to sharing λ-terms across the syntax boundary. In practice the PMA mechanism requires two more entry points: one for setting up the communication between the protected and unprotected memory and one that handles callbacks from the unprotected memory to the protected memory. Because every program will have these two entry points, they do not affect contextual equivalence and we thus do not model them.

3.4.3 Simplified Function Calls

Instead of wrapping shared λ-terms into a λ-term of the receiving language as in the λ_m-calculus, the combined calculi are extended with terms that denote the availability of λ-terms from the opposing side. In the λ_a-calculus that term is as mentioned previously a name $\mathsf{n_i}$, in the λ_s-calculus that term is $\boldsymbol{\sigma}\mathbf{S}\mathsf{A}\,(\lambda\mathsf{x.t})$. This term models the direct access that the λ_s-calculus has to the functions of the λ_a-calculus.

When a λ_s-calculus program now calls a λ_a-calculus function $(\lambda\mathsf{x.t})$, for example, it simply passes a reference to that function as well as its arguments to the λ_a-calculus program on the other side of the syntactical boundary.

This is in contrast to the λ_m-calculus, where calling the λ_a-calculus function $(\lambda\mathsf{x.t})$ is done by calling an insecure memory chunk that encodes an application of that function to a callback to the λ_s-calculus.

4. The λ^+-Calculus

To resolve the modeling limitations of the λ_m-calculus we introduce a new combined language the λ^+-calculus that incorporates the solutions proposed in Section 3.4.

The λ^+-calculus introduces new syntax (Section 4.1), a larger program definition (Section 4.2), new operational semantics (Section 4.3), additional typing rules (Section 4.4) and a modified notion of type soundness (Section 4.5). We

$\mathsf{t} ::= \mathsf{x}$	$\mathsf{v} ::= \mathsf{b}$	$\mathsf{b} ::= \#\mathsf{t}$	$\mathsf{E} ::= [\cdot]$
$\mid \mathsf{t}_1\ \mathsf{t}_2$	$\mid (\lambda \mathsf{x}.\mathsf{t})$	$\mid \#\mathsf{f}$	$\mid \mathsf{E}\ \mathsf{t}$
$\mid \mathsf{t}_1 \equiv \mathsf{t}_2$	$\mid \mathsf{wrong}$		$\mid \mathsf{v}\ \mathsf{E}$
$\mid \mathsf{call}\ \mathsf{n_i}\ \mathsf{t}$	$\mid \mathsf{n_i}$		$\mid \mathsf{call}\ \mathsf{n_i}\ \mathsf{E}$
$\mid \mathsf{v}$	$\mid \mathsf{call}\ \mathsf{n_i}\ \mathsf{v}$		
$\mathbf{t} ::= \mathbf{x}$	$\mathbf{v} ::= \mathbf{b}$	$\boldsymbol{\sigma} ::= \mathbf{Bool}$	$\mathbf{E} ::= [\cdot]$
$\mid \mathbf{t_1}\ \mathbf{t_2}$	$\mid (\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t})$	$\mid \boldsymbol{\sigma} \rightarrow \boldsymbol{\sigma}$	$\mid \mathbf{E}\ \mathbf{t}$
$\mid \mathbf{if}\ \mathbf{t_1}\ \mathbf{t_2}\ \mathbf{t_3}$	$\mid {}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\mathsf{t})$	$\mathbf{b} ::= \#\mathbf{t}$	$\mid \mathbf{v}\ \mathbf{E}$
$\mid \mathbf{v}$		$\mid \#\mathbf{f}$	$\mid \mathbf{if}\ \mathbf{E}\ \mathbf{t}\ \mathbf{t}$

Figure 2: The λ^+-calculus adds names and direct function references to the syntax.

provide a few examples to illustrate the effectiveness of our attacker model in this composed language (Section 4.6).

4.1 Syntax

The syntax of the λ^+-calculus combines the λ_s-calculus and λ_a-calculus as illustrated in Figure 2.

Felleisen-and-Hieb-style reduction semantics are used to specify the operational semantics [6]. To that end the λ_s-calculus evaluation context $\mathbf{E}$ and the λ_a-calculus evaluation context E are introduced to lift the basic reduction steps to a standard left-to-right call-by-value semantics.

The λ_s-calculus is extended with a term: ${}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\mathsf{t})$ that denotes the direct access programs in the λ_s-calculus have to the functions of a λ_a-calculus program, as those functions reside in the unprotected memory. This term is type annotated to enable the semantics to wrap the call to the denoted function with a typecheck on the output of that function.

Terms of the λ_a-calculus cannot directly access the protected terms of the λ_s-calculus. They are instead limited to the designated entry-points of the PMA mechanism. The names $\mathsf{n_i}$ model these designated entry-points. The set of names are denumerable as: $\mathsf{n_i} \neq \mathsf{n_j}$ if $i \neq j$. A λ_a-calculus attacker can compare these names through the syntactical equality operator $\equiv$ and can apply them to values using the term $\mathsf{call}\ \mathsf{n_i}\ \mathsf{v}$.

The λ_a-calculus is not extended with any means to create new names dynamically. Instead we assume that, as in the example in Section 3.4.2, the attacker guesses the deterministically created names beforehand. For every possible name $\mathsf{n_i}$ we have that there exists a context C that can distinguish it from other names ($\mathsf{C} = (\mathsf{n_i} \equiv [\cdot])$) or apply it to some value v ($\mathsf{C} = (\mathsf{call}\ \mathsf{n_i}\ \mathsf{v})$). The ability to create names dynamically does not produce stronger contexts.

Note that the λ_s-calculus does not treat *call* terms as values. This is because, as mentioned in Section 3.4.1, the λ_s-calculus is extended with a call stack capable of encoding such calls and returns. The λ_a-calculus is, however, an attacker and should thus be able to manipulate the structure of its call stack.

4.2 Program definition

The λ^+-calculus does not syntactically embed the composed languages λ_s- and λ_a-calculus. The calculus instead combines a λ_s-calculus configuration, that is assumed to reside in secure memory, with a λ_a-calculus configuration, that is assumed to reside in the unprotected memory, into the program definition.

The program definition considers two modes of interaction. In one mode the λ_s-calculus configuration is executing, while the λ_a-calculus configuration is waiting on input from the λ_s-calculus configuration. In the other mode a λ_a-calculus configuration is executing, while the λ_s-calculus configuration is either empty or waiting on a callback from the λ_a-calculus configuration.

The λ_s Configuration A λ_s-calculus configuration $\mathbf{S}$ is defined as follows:

$$\begin{aligned} \mathbf{S} = \mathbf{N} \Vdash \boldsymbol{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma} \quad &\mid \quad \mathbf{N} \Vdash \boldsymbol{\Sigma} \\ \textit{where} \quad \boldsymbol{\Sigma} &= \overline{\mathbf{E}} : \overline{\boldsymbol{\sigma}_\mathbf{f}} \ \mid \ \boldsymbol{\varepsilon} \\ \textit{and} \quad \mathbf{N} &::= \star \mid \mathbf{N}', \mathsf{n_i} \mapsto (\mathbf{t}, \boldsymbol{\sigma}) \end{aligned}$$

where $\overline{\mathbf{E}}$ denotes a sequence of open evaluation contexts $\mathbf{E}$ with a hole $[\cdot]$ and $\overline{\boldsymbol{\sigma}_\mathbf{f}}$ denotes a sequence of function types $\boldsymbol{\sigma_1} \rightarrow \boldsymbol{\sigma_2}$. The sequence of type annotated open evaluation contexts $\boldsymbol{\Sigma}$ thus represents the λ_s-calculus program's view of the evaluation stack in a way that is similar to the stack mechanism used by Jeffrey and Rathke's fully abstract trace semantics for Java Jr. [14].

The map $\mathbf{N}$ is used to keep track of the names that a λ_s-calculus program shares with a program in the λ_a-calculus.

The first case of the definition describes an executing λ_s-configuration, denoted as $\mathbf{S_e}$, that executes a λ_s-term $\mathbf{t}$. This λ_s-term is type annotated to allow the operational semantics to construct new typed annotated evaluation stacks at runtime. The second case describes a passive λ_s-configuration, denoted as $\mathbf{S_p}$, that waits on correctly typed input from the λ_a-calculus term.

The λ_a Configuration A λ_a-calculus configuration A is defined as follows:

$$\mathsf{A} ::= \overline{\mathsf{C}} \bullet \mathsf{t} \ \mid \ \overline{\mathsf{C}}$$

where $\overline{\mathsf{C}}$ denotes a sequence of λ_a-contexts C. A context C differs from an open evaluation context E: the former is any term with a hole in it, the latter is any term with a hole in the place where the next reduction step happens. We thus define a λ_a-calculus program's view of the evaluation stack as a sequence of possible attacks.

The first case of the definition describes an executing λ_a-configuration, denoted as A_e. The second case describes a passive λ_a-configuration that awaits input from a λ_s-configuration $\mathbf{S_e}$, denoted as A_p.

Internal Computations of the $\lambda_{\mathbf{a}}$-Calculus

$$\overline{\mathsf{C}} \bullet \mathsf{E}[((\lambda \mathsf{x.t})\ \mathsf{v})] \mid\mid \mathbf{S_p} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{E}[\mathsf{t}[\mathsf{v}/\mathsf{x}]] \mid\mid \mathbf{S_p} \quad \text{(A-App)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{E}[(\mathsf{v}_1\ \mathsf{v}_2)] \mid\mid \mathbf{S_p} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{E}[\mathsf{wrong}] \mid\mid \mathbf{S_p} \quad \textit{where } \mathsf{v}_1 \neq (\lambda \mathsf{x.t}) \quad \text{(A-ApW)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{E}[\mathsf{t}_1 \equiv \mathsf{t}_2] \mid\mid \mathbf{S_p} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{E}[\#\mathsf{t}] \mid\mid \mathbf{S_p} \quad \textit{where } \mathsf{t}_1 = \mathsf{t}_2 \quad \text{(A-Eq1)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{E}[\mathsf{t}_1 \equiv \mathsf{t}_2] \mid\mid \mathbf{S_p} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{E}[\#\mathsf{f}] \mid\mid \mathbf{S_p} \quad \textit{where } \mathsf{t}_1 \neq \mathsf{t}_2 \quad \text{(A-Eq2)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{E}[\mathsf{wrong}] \mid\mid \mathbf{S_p} \twoheadrightarrow \mathsf{wrong} \mid\mid \mathbf{S_p} \quad \text{(A-Wr)}$$

Internal Computations of the $\lambda_{\mathbf{s}}$-Calculus

$$\overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[((\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t})\ \mathbf{v})] : \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{t}[\mathbf{v}/\mathbf{x}]] : \boldsymbol{\sigma} \quad \text{(S-App)}$$

$$\overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{if}\ \#\mathbf{t}\ \mathbf{t_2}\ \mathbf{t_3}] : \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{t_2}] : \boldsymbol{\sigma} \quad \text{(S-IFT)}$$

$$\overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{if}\ \#\mathbf{f}\ \mathbf{t_2}\ \mathbf{t_3}] : \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{t_3}] : \boldsymbol{\sigma} \quad \text{(S-IFF)}$$

Protected Computations

$$\overline{\mathsf{C}}, \mathsf{C} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\lambda \mathbf{x} : \boldsymbol{\sigma}'.\mathbf{t}) : \boldsymbol{\sigma}_f \twoheadrightarrow \overline{\mathsf{C}}, \mathsf{C}[\mathsf{n_i}] \mid\mid \mathbf{N}, \mathsf{n_i} \mapsto ((\lambda \mathbf{x}.\mathbf{t}), \boldsymbol{\sigma}) \Vdash \mathbf{\Sigma} \quad \textit{where } i = |\mathbf{N}| + 1 \quad \text{(S-Name)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{n_i} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{t}] : \boldsymbol{\sigma}_2 \quad \textit{where } \mathbf{N}(\mathsf{n_i}) = (\mathbf{t}, \boldsymbol{\sigma_1}) \quad \text{(A-Name)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \bullet \mathsf{n_i} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \twoheadrightarrow \overline{\mathsf{C}}, \mathsf{C}[\mathsf{wrong}] \mid\mid \star \Vdash \boldsymbol{\varepsilon} \quad \textit{where } \mathbf{N}(\mathsf{n_i}) \neq (\mathbf{t}, \boldsymbol{\sigma}) \textit{ or } \mathsf{n_i} \notin \mathit{dom}(\mathbf{N}) \quad \text{(WrongN)}$$

$$\overline{\mathsf{C}} \bullet \mathsf{call}\ \mathsf{n_i}\ \mathsf{v} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{v} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ [\cdot]) : \boldsymbol{\sigma} \quad \textit{where } \mathbf{N}(\mathsf{n_i}) = (\mathbf{t}, \boldsymbol{\sigma}) \quad \text{(A-Call)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \bullet \mathsf{call}\ \mathsf{n_i}\ \mathsf{v} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{C}[\mathsf{wrong}] \mid\mid \star \Vdash \boldsymbol{\varepsilon} \quad \textit{where } \mathsf{n_i} \notin \mathit{dom}(\mathbf{N}) \quad \text{(WrongC)}$$

Unprotected Computations

$$\overline{\mathsf{C}} \bullet (\lambda \mathsf{x.t}) \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_f \to \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[(\lambda \mathbf{y} : \boldsymbol{\sigma_1}.((^{\boldsymbol{\sigma_f}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t}))\ \mathbf{y}))] : \boldsymbol{\sigma} \quad \text{(A-Lam)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \bullet \mathsf{v} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_f \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{C}[\mathsf{wrong}] \mid\mid \star \Vdash \boldsymbol{\varepsilon} \quad \textit{where } \mathsf{v} \neq (\lambda \mathsf{x.t}) \quad \text{(WrongL)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet {}^{\boldsymbol{\sigma_f}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t}) : \boldsymbol{\sigma}_f \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{C}[(\lambda \mathsf{x.t})] \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \quad \text{(S-Lam)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[((^{\boldsymbol{\sigma_f}}\mathbf{S}\mathsf{A}\ \mathsf{t})\ \mathbf{v})] : \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}}, \mathsf{C}[(\mathsf{t}\ [\cdot])] \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma} \bullet \mathbf{v} : \boldsymbol{\sigma}_1 \quad \text{(S-Call)}$$

Value Passing

$$\overline{\mathsf{C}} \bullet \mathsf{b} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \mathbf{Bool} \to \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{b}] : \boldsymbol{\sigma} \quad \text{(A-Bool)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{b} : \mathbf{Bool} \twoheadrightarrow \overline{\mathsf{C}} \bullet \mathsf{C}[\mathsf{b}] \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \quad \text{(S-Bool)}$$

$$\overline{\mathsf{C}}, \mathsf{C} \bullet \mathsf{v} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \mathbf{Bool} \to \boldsymbol{\sigma} \twoheadrightarrow \overline{\mathsf{C}}, \mathsf{C}[\mathsf{wrong}] \mid\mid \star \Vdash \boldsymbol{\varepsilon} \quad \textit{where } \mathsf{v} \neq \mathsf{b} \quad \text{(WrongB)}$$

Figure 3: The reduction rules of the λ^+-calculus. The type $\boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2$ is shortened to $\boldsymbol{\sigma}_f$ for the sake of brevity. The expression $\mathbf{E}[\mathbf{t}]$ fills the hole of $\mathbf{E}$ with $\mathbf{t}$ in the obvious way and the expression $\mathsf{C}[\mathsf{t}]$ fills the hole of C in the obvious way.

The Program A program P that considers the two possible execution states is defined as follows:

$$P ::= \mathsf{t} \mid\mid \mathbf{S_p} \ \ or \ \ \overline{\mathsf{C}} \mid\mid \mathbf{S_e}$$

where $\mid\mid$ separates the configurations of the λ_a- and λ_s-calculi and by extension divides the unprotected memory from the protected memory.

To simplify the later full abstraction results we assume that the secure λ_s-calculus configuration is always first to execute: $P_0 = \overline{\mathsf{C}} \mid\mid \mathbf{S_e}$. A program thus always terminates with a value on the insecure side: $P_f = \overline{\mathsf{C}} \bullet \mathsf{v} \mid\mid \mathbf{N} \Vdash \boldsymbol{\varepsilon}$.

4.3 Operational Semantics

The reduction rules of the λ^+-calculus, denoted as $P \twoheadrightarrow P'$, are illustrated in Figure 3. We divide these rules into four categories: internal computations, protected computations, unprotected computations and primitive value passing.

Internal Computations Internal computations are reduction rules that only affect the terms of one of the two languages. In these reduction rules the terms of one of the languages remain unchanged. Function application, for example, is an internal computation (rules A-App and S-App).

Note that the rule A-Wr artificially restricts the attacker model. In practice, an arbitrary attacker can recover from errors. However, the λ_s-configuration is cleared out whenever the λ_a-attacker returns an incorrectly typed value (rules WrongN, WrongC, WrongL and WrongB). As such the actions of an attacker after something has gone wrong are not relevant to the security result of this work.

Protected Computations Protected computations are reduction rules that enable the λ_a-calculus terms to call functions of the λ_s-calculus. In these reduction rules the map $\mathbf{N}$ ensures that a λ_a-calculus attacker is limited to the λ_s-terms that have been shared with it.

In rule S-Name a λ_s-calculus λ-term is passed across to the hole of the λ_a-context, by sharing the next name $\mathsf{n_i}$ from the countable set of names with the λ_a-configuration. This new name and its associated type and term are stored in the map $\mathbf{N}$.

In rules A-Name and WrongN, a name $\mathsf{n_i}$ is passed back to the head of the λ_s-calculus evaluation stack $\mathbf{\Sigma}$. If the name is associated with a λ_s-term that has the same type as the hole then that λ_s-term fills the hole, otherwise the system terminates in error.

In rules A-Call and WrongC a name $\mathsf{n_i}$ is applied to a value. If the name matches a λ_s-term in the map $\mathbf{N}$, a new λ_s-calculus reduction context, that applies a λ_a-value to the fetched term, is pushed onto the λ_s-calculus evaluation stack.

Unprotected Computations Unprotected computations are reduction rules that enable the λ_s-calculus to call functions of the λ_a-calculus. In these reduction rules only the type annotations and associated dynamic type checks are required to ensure security.

In rules A-Lam and WrongL a λ_a-calculus λ-term is passed across to the head of the λ_s-calculus evaluation stack $\mathbf{\Sigma}$ if it expects a function. As a result the transmitted λ-term is tagged with the type of the hole it fills.

In rule S-Lam, a λ_a-calculus λ-term is passed back to the context C, the head of the λ_a-configurations sequence of possible attacks.

In rule S-Call the λ_a-calculus function ${}^{\boldsymbol{\sigma_f}}\mathbf{S}\mathsf{A}\,\mathsf{t}$ is applied to a λ_s-calculus value. The resulting reduction step differs from the reduction step A-Call in two ways. The first difference is that the rule does not push a new reduction context onto the λ_a-calculus. Instead the function call is directly inserted into the waiting λ_a-context. This direct approach allows us to accurately model the observational capabilities of an arbitrary machine level attacker, as such an attacker can directly observe any call to its functions. Note that as a result of this direct approach, there is no guarantee that the function call to ${}^{\boldsymbol{\sigma_f}}\mathbf{S}\mathsf{A}\,\mathsf{t}$ will be succesful. Given that the λ_a-calculus models the arbitrary attacker this is to be accepted.

The second difference is that in the λ_s-calculus a cross boundary function call is not a value. In the λ_a-calculus cross boundary function calls are values, to ensure that every such call can be followed up by an attack C.

Primitive Value Passing Passing primitive value does not result in the creation of new names or stack frames. In reduction rules A-Bool and S-Bool the booleans are converted to the appropriate representation.

4.4 Typing rules

The simply typed call-by-value lambda calculus (the λ_s-calculus) is typed as follows:

$$\frac{\mathbf{x} : \boldsymbol{\sigma} \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash \mathbf{x} : \boldsymbol{\sigma}} \qquad \frac{\mathbf{\Gamma}, \mathbf{x} : \boldsymbol{\sigma_1} \vdash \mathbf{t} : \boldsymbol{\sigma_2}}{\mathbf{\Gamma} \vdash (\lambda \mathbf{x} : \boldsymbol{\sigma_1}.\mathbf{t}) : \boldsymbol{\sigma_1} \to \boldsymbol{\sigma_2}}$$

$$\frac{}{\mathbf{b} : \mathbf{Bool}} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{t_1} : \boldsymbol{\sigma_1} \to \boldsymbol{\sigma_2} \qquad \mathbf{\Gamma} \vdash \mathbf{t_2} : \boldsymbol{\sigma_1}}{\mathbf{\Gamma} \vdash \mathbf{t_1}\ \mathbf{t_2} : \boldsymbol{\sigma_2}}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{t_1} : \mathbf{Bool} \qquad \mathbf{\Gamma} \vdash \mathbf{t_2} : \boldsymbol{\sigma} \qquad \mathbf{\Gamma} \vdash \mathbf{t_3} : \boldsymbol{\sigma}}{\mathbf{\Gamma} \vdash \mathbf{if}\ \mathbf{t_1}\ \mathbf{t_2}\ \mathbf{t_3} : \boldsymbol{\sigma}}$$

The λ^+-calculus extends the λ_s-calculus with a value ${}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\,(\lambda \mathsf{x.t})$ which is typed as follows:

$$\frac{}{\mathbf{\Gamma} \vdash {}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\,(\lambda \mathsf{x.t}) : \boldsymbol{\sigma}}$$

To type a Program P we only type the configuration $\mathbf{S}$ using the following rules.

$$\frac{\mathbf{\Gamma} \vdash \mathbf{S}}{\mathbf{\Gamma} \vdash \mathsf{A} \mid\mid \mathbf{S}} \qquad \frac{}{\mathbf{\Gamma} \vdash \star} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{N} \qquad \mathbf{\Gamma} \vdash (\lambda \mathbf{x.t}) : \boldsymbol{\sigma}}{\mathbf{\Gamma} \vdash \mathbf{N}, [\mathsf{n_i} \mapsto ((\lambda \mathbf{x.t}), \boldsymbol{\sigma})]}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{N}}{\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \varepsilon} \qquad \frac{\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{\Sigma} \qquad \mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{t} : \boldsymbol{\sigma}}{\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma}}$$

$$\frac{\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{\Sigma} \qquad \mathbf{\Gamma}, \mathbf{x} : \boldsymbol{\sigma_1} \vdash \mathbf{N} \Vdash \mathbf{E}[\mathbf{x}] : \boldsymbol{\sigma_2}}{\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma_1} \to \boldsymbol{\sigma_2}}$$

To type check a configuration $\mathbf{S}$, each individual reduction context of the evaluation stack $\mathbf{\Sigma}$ is type checked by adding the type of the hole to the variable scope. Each association in the map $\mathbf{N}$ is typed as well.

4.5 Type Soundness

As mentioned previously only the λ_s-configuration of a program P is type checked. As such we cannot introduce a traditional notion of type soundness. Instead we establish that whenever a program gets stuck or reduces to the error wrong, the λ_a-configuration is the cause.

Note that this approach to type soundness is very similar to Wadler's and Findler's blame calculus [24]. The similarities with their work are, however, out of scope for this paper.

Theorem 1 (Type Preservation)**.** *Given* $\mathbf{\Gamma} \vdash P$ *and* $P \twoheadrightarrow P'$ *we have that* $\mathbf{\Gamma} \vdash P'$*.*

Proof Sketch. We consider the most important cases:

- $\mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma} \twoheadrightarrow \mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t}' : \boldsymbol{\sigma}$: We have that $\mathbf{\Gamma} \vdash \mathbf{t}' : \boldsymbol{\sigma}$ follows from the fact that the λ^+-calculus preserves the reduction rules of the λ_s-calculus.
- $\mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma} \twoheadrightarrow \mathsf{A}' \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t}' : \boldsymbol{\sigma}$: The internal reduction rules of the λ_a-calculus do not modify the λ_s-configuration
- $\mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \twoheadrightarrow \mathsf{A}' \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E}[\mathbf{t}] : \boldsymbol{\sigma}_2$: By the reduction rules WrongN, WrongL, WrongC and WrongB we have that: $\mathbf{\Gamma} \vdash \mathbf{t} : \boldsymbol{\sigma}_1$ and thus that $\mathbf{\Gamma} \vdash \mathbf{E}[\mathbf{t}] : \boldsymbol{\sigma}_2$.
- For rule A-Call we have that $\mathbf{\Gamma} \vdash \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ [\cdot]) : \boldsymbol{\sigma}$ as $\mathbf{N}(\mathsf{n_i}) = (\mathbf{t}, \boldsymbol{\sigma})$ and $\mathbf{\Gamma} \vdash \mathbf{N}$ by the assumption.

□

Theorem 2 (Type Progress)**.** *Given* $\mathbf{\Gamma} \vdash \mathbf{S}$ *then if* $P \not\twoheadrightarrow P'$ *or* $P \twoheadrightarrow \mathsf{wrong} \mid\mid \star \Vdash \boldsymbol{\varepsilon}$ *then* A *is the cause.*

Proof Sketch. We consider the most relevant cases:

- $P = \mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma}$: We have that $P \twoheadrightarrow \mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t}' : \boldsymbol{\sigma}$ as the λ^+-calculus preserves the reduction rules of the λ_s-calculus.
- $P = \mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{v} : \boldsymbol{\sigma}$: By the reduction rules S-Name, S-Lam and S-Bool we have that $P \twoheadrightarrow P'$ if and only if there is at least one context C in the passive λ_a-configuration $\overline{\mathsf{C}}$.
- $P = \mathsf{A} \mid\mid \mathbf{N} \Vdash \mathbf{\Sigma}$: We have that $P \twoheadrightarrow P'$ if the executing λ_a-configuration t reduces to a value v. If the value is not of the correct type we have that $P' = \mathsf{wrong} \mid\mid \star \Vdash \boldsymbol{\varepsilon}$.

□

4.6 Attacker Model Examples

We illustrate the effectiveness of the attacker model in the combined λ^+-calculus by detailing λ_a-configurations that can distinguish between two λ_s-configurations that perform different function calls and two λ_s-configurations with different maps $\mathbf{N}$.

Different Function Calls Consider the following two λ_s-configurations.

$$\mathbf{S}_1 = \star \Vdash \mathbf{if}[\cdot](^{\sigma}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\#\mathsf{t})\ \#\mathbf{t})\#\mathbf{f} : \mathbf{Bool} \to \mathbf{Bool} \bullet (^{\sigma}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\#\mathsf{t})\ \#\mathbf{t}) : \mathbf{Bool}$$

$$\mathbf{S}_2 = \star \Vdash \mathbf{if}[\cdot](^{\sigma}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\#\mathsf{t})\ \#\mathbf{f})\#\mathbf{f} : \mathbf{Bool} \to \mathbf{Bool} \bullet (^{\sigma}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x}.\#\mathsf{t})\ \#\mathbf{t}) : \mathbf{Bool}$$

These two λ_s-configurations call the same λ_a-function twice. In the second function call, however, the function is applied to $\#\mathsf{t}$ in $\mathbf{S}_1$ and $\#\mathsf{f}$ in $\mathbf{S}_2$. The following λ_a-configuration can distinguish between the two.

$$\mathsf{A_a} = (((\lambda \mathsf{x}.\#\mathsf{t})\ \#\mathsf{t}) \equiv [\cdot]), [\cdot]$$

The first context of $\mathsf{A_a}$ will process the first function call as expected by the λ_s-calculus. The second context compares the contents of the second function call.

Different Maps The following two λ_s-configurations:

$$\mathbf{S}_1 = [\mathsf{n}_1 \mapsto ((\lambda \mathbf{x}.\#\mathbf{t}), \boldsymbol{\sigma})], [\mathsf{n}_2 \mapsto ((\lambda \mathbf{x}.(\lambda \mathbf{y}.\mathbf{y})), \boldsymbol{\sigma}')] \Vdash \boldsymbol{\varepsilon}$$

$$\mathbf{S}_2 = [\mathsf{n}_1 \mapsto ((\lambda \mathbf{x}.\#\mathbf{t}), \boldsymbol{\sigma})], [\mathsf{n}_2 \mapsto ((\lambda \mathbf{x}.\#\mathbf{t}), \boldsymbol{\sigma})] \Vdash \boldsymbol{\varepsilon}$$

differ only for the name n_2, which points to two non-contextually equivalent functions. The following λ_a-config. can distinguish between the two.

$$\mathsf{A_a} = (\mathsf{n}_3 \equiv [\cdot]) \bullet \mathsf{call}\ \mathsf{n}_2\ \#\mathsf{f}$$

This configuration first calls the name n_2 with a value $\#\mathsf{f}$. As a result it will receive a new name n_3 in one case and a value $\#\mathsf{t}$ in the other the attacker context will thus reduce to $\#\mathsf{t}$ in the first case and $\#\mathsf{f}$ in the second.

5. Full Abstraction

Proving that the λ^+-calculus is capable of preserving the abstractions of the λ_s-calculus is done by introducing a fully abstract compilation scheme form the λ_s-calculus to the λ^+-calculus. This fully abstract compilation scheme preserves the equivalences of the λ_s-calculus in the λ^+-calculus.

Direct proofs over contextual equivalence are difficult however, as one needs to reason about any reduction in any context. To that end we develop notions of bisimulation that coincide with contextual equivalence for the λ_s-calculus (Section 5.1) and for the compiled terms in the λ^+-calculus (Section 5.2). Proving that the compilation scheme is fully abstract is done by relating the bisimulations (Section 5.3).

5.1 A Congruent Bisimulation for the λ_s-Calculus

We define contextual equivalence $\simeq_s$ (Section 5.1.1) and a bisimulation relation $\mathcal{S}$ (Section 5.1.2) and over the terms of the λ_s-calculus. The bisimulation is a congruence and thus coincides with contextual equivalence (Section 5.1.3).

5.1.1 Contextual Equivalence for the λ_s-Calculus

A λ_s context $\mathbf{C}$ is a λ_s-term with a single hole $[\cdot]$ in it. We now define contextual equivalence as follows [5]:

Definition 1. Contextual equivalence ($\simeq_s$) *is defined as:*
$\mathbf{t_1} \simeq_s \mathbf{t_2} \iff \forall \mathbf{C}.\ \exists \mathbf{b}.\ \mathbf{C}[\mathbf{t_1}] \to^* \mathbf{b} \iff \mathbf{C}[\mathbf{t_2}] \to^* \mathbf{b}$

where $\to^*$ denotes convergence.

Note that two contextually equivalent terms are the same type as a context observes the same typing rules as the programs of the λ_s-calculus.

5.1.2 A Bisimulation for the λ_s-Calculus

There have been multiple different bisimulations over the simply typed λ-calculus. In this paper we adopt Gordon's definition and associated LTS over the simply typed λ-calculus [10]. The LTS models the interaction between the context and a λ_s-calculus term.

The LTS is a triple $(\mathbf{t}, \boldsymbol{\alpha}, \xrightarrow{\boldsymbol{\alpha}})$ where terms $\mathbf{t}$ are the states, $\boldsymbol{\alpha}$ the set of labels and $\xrightarrow{\boldsymbol{\alpha}}$ the labelled transitions

between states. The labels $\boldsymbol{\alpha}$ are defined as follows:

$$\begin{aligned}\boldsymbol{\alpha} &::= \boldsymbol{\gamma} \mid \boldsymbol{\tau} \\ \boldsymbol{\gamma} &::= @\mathbf{v} \mid \mathbf{true} \mid \mathbf{false}\end{aligned}$$

Labelled Reductions are of the form $\mathbf{t} \xrightarrow{\gamma} \mathbf{t}'$. Values are observable by a context and are labelled as follows:

$$(\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t}) \xrightarrow{@\mathbf{v}} ((\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t})\ \mathbf{v}) \quad \text{(Label-App)}$$
$$\textit{where} \vdash @\mathbf{v} : \boldsymbol{\sigma}$$
$$\#\mathbf{t} \xrightarrow{\mathbf{true}} \#\mathbf{t} \quad \text{(Label-T)}$$
$$\#\mathbf{f} \xrightarrow{\mathbf{false}} \#\mathbf{f} \quad \text{(Label-F)}$$

The LTS models the interactions between a λ_s-calculus context $\mathbf{C}$ and a λ_s-calculus program. In reduction rule Label-App the context observes the contents of a λ-term by applying values to it. Because the λ_s-calculus terms and contexts are typed the applied values are restricted to those that conform to the type rules.

In reduction rules Label-T and Label-F boolean values are observed directly as booleans contain no abstractions.

Reduction steps between terms cannot be observed by a context and are thus labelled as silent:

$$\frac{\mathbf{t} \to \mathbf{t}'}{\mathbf{t} \xrightarrow{\tau} \mathbf{t}'} \quad \text{(Label-S)}$$

We define a weak bisimulation over this LTS. In contrast to a strong bisimulation, such a bisimulation does not use the silent transitions between two states. Define the transition relation $\mathtt{t} \overset{\gamma}{\Rightarrow} \mathtt{t}'$ as $\mathtt{t} \xrightarrow{\tau}{}^{*} \xrightarrow{\gamma} \mathtt{t}'$ where $\xrightarrow{\tau}{}^{*}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau}$.

Bisimulation is now defined as follows:

Definition 2. *The relation $\mathcal{S}$ is a **bisimulation** if and only if $t_1\ \mathcal{S}\ t_2$ implies:*

(1) Given $t_1 \overset{\gamma}{\Rightarrow} t_1'$ there is $t_2' : t_2 \overset{\gamma}{\Rightarrow} t_2'\ \wedge\ t_1'\ \mathcal{S}\ t_2'$

(2) Given $t_2 \overset{\gamma}{\Rightarrow} t_2'$ there is $t_1' : t_1 \overset{\gamma}{\Rightarrow} t_1'\ \wedge\ t_1'\ \mathcal{S}\ t_2'$

We denote bisimilarity, the largest bisimulation, as $\approx_s$.

5.1.3 Full Abstraction of the Bisimilarity

We conclude that the bisimilarity $\approx_s$ coincides with contextual equivalence $\simeq_s$.

Theorem 3 (Full Abstraction of the Bisimilarity)**.**
$\mathbf{t_1} \simeq_s \mathbf{t_2} \Leftrightarrow \mathbf{t_1} \approx_s \mathbf{t_2}$

Due to space constraints, the proof of this theorem and those of the following lemma have been placed in an online Appendix A[1].

A proof sketch of this theorem is available in Appendix A.1. The proof sketch is a straight forward adaptation of Gordon's proof of congruence for a bisimulation over PCF [10]. The proof sketch leverages the symmetry properties of both bisimilarity and contextual equivalence and splits the theorem into two sublemma: contextual equivalence implies bisimilarity (Completeness) and bisimilarity implies contextual equivalence (Soundness). The latter is established by applying Howe's method [11].

[1] https://dl.dropboxusercontent.com/u/14314349/operational/proofs.pdf

5.2 A Congruent Bisimulation for the λ^+-Calculus

As in Section 5.1 we define a notion of bisimulation, whose bisimilarity is a congruence over the terms of the λ^+-calculus (Section 5.2.2). We first introduce a definition of contextual equivalence that reflects the assumptions of the compilation scheme (Section 5.2.1).

5.2.1 Contextual Equivalence for the λ^+-Calculus

The goal of the secure compilation scheme is to preserve the abstractions of the λ_s-calculus in the combined λ^+-calculus. The secure compilation does not simply compile λ_s-calculus terms into a λ_s- or λ_a-calculi context. Instead it produces a λ_s configuration $\mathbf{S}$ that interoperates with any possible λ_a-configuration in a secure manner. We can thus define contextual equivalence as follows:

Definition 3. (Contextual equivalence for λ_s Configurations) $\mathbf{S}_1 \simeq_c \mathbf{S}_2 \iff \forall \mathsf{A}.(\mathsf{A} \mid\mid \mathbf{S}_1)\Uparrow \iff (\mathsf{A} \mid\mid \mathbf{S}_2)\Uparrow$

where $\Uparrow$ denotes divergence [20]. A program P diverges if it executes an unbounded number of reduction steps. Formally $P\Uparrow \iff \forall n \in \mathbb{N}.\exists P'.\mathbf{t} \to^n P'$.

Note that this formulation of contextual equivalence differs from the definition in Section 5.1.1. We adopt this divergence based definition of contextual equivalence to simplify the proof of congruence.

5.2.2 A Bisimulation for the λ^+-Calculus

The LTS is a triple $(\mathbf{S}, \alpha^+, \xrightarrow{\alpha^+})$ where configurations $\mathbf{S}$ are the states, α^+ the set of labels and $\xrightarrow{\alpha^+}$ the labelled transitions between states. The labels α^+ are defined as follows:

$$\begin{aligned}\alpha^+ &::= \gamma^+ \mid \tau^+ \\ \gamma^+ &::= (\lambda \mathsf{x.t}) \mid @\mathsf{v} \mid \mathsf{n_i} \mid \gg (\lambda \mathsf{x.t})\ ;\ \mathsf{v} \mid \gg \mathsf{n}\ ;\ \mathsf{v} \\ &\quad\ \mathsf{true} \mid \mathsf{false} \mid \mathsf{wrong} \mid \mathsf{done}\end{aligned}$$

These labels describe what a λ_a-configuration A observes from its interactions with a configuration $\mathbf{S}$. The labelled reductions of the LTS (Figure 4) are of the form $\mathbf{S} \xrightarrow{\gamma^+} \mathbf{S}'$. While the λ_a-configuration is not represented in these labelled reductions, the changes to the λ_a-configuration can be derived from the labels.

The labelled reduction rules O-True, O-False, O-Name and O-Lambda describe the values that a λ_a-configuration observes from a λ_s-configuration. The labelled reduction rules I-Bool, I-Name, and I-Lambda describe the values a

$$\mathbf{N} \Vdash \mathbf{\Sigma} \bullet \#\mathbf{f} : \mathbf{Bool} \xrightarrow{\mathsf{false}} \mathbf{N} \Vdash \mathbf{\Sigma} \quad \text{(O-False)} \qquad \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \#\mathbf{t} : \mathbf{Bool} \xrightarrow{\mathsf{true}} \mathbf{N} \Vdash \mathbf{\Sigma} \quad \text{(O-True)}$$

$$\star \Vdash \varepsilon \xrightarrow{\mathsf{done}} \star \Vdash \varepsilon \quad \text{(O-Done)} \qquad \mathbf{N} \Vdash \mathbf{\Sigma} \bullet {}^{\boldsymbol{\sigma}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t}) : \boldsymbol{\sigma} \xrightarrow{(\lambda \mathsf{x.t})} \mathbf{N} \Vdash \mathbf{\Sigma} \quad \text{(O-Lambda)}$$

$$\mathbf{N} \Vdash \mathbf{\Sigma} \xrightarrow{\mathsf{wrong}} \star \Vdash \varepsilon \quad \text{(I-Wrong)}$$

$$\mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \mathbf{Bool} \to \boldsymbol{\sigma} \xrightarrow{@\mathsf{b}} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{b}] : \boldsymbol{\sigma} \quad \text{(I-Bool)}$$

$$\mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t}) : \boldsymbol{\sigma}_f \xrightarrow{\mathsf{n_i}} \mathbf{N}, \mathsf{n_i} \mapsto ((\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t}), \boldsymbol{\sigma}_\mathbf{f}) \Vdash \mathbf{\Sigma} \quad \text{(O-Name)}$$

$$\mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_f \to \boldsymbol{\sigma} \xrightarrow{@\mathsf{n_i}} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[\mathbf{t}] : \boldsymbol{\sigma} \quad \textit{where } \mathbf{N}(\mathsf{n_i}) = (\mathbf{t}, \boldsymbol{\sigma}_f) \quad \text{(I-Name)}$$

$$\mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_f \to \boldsymbol{\sigma} \xrightarrow{@(\lambda \mathsf{x.t})} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[{}^{\boldsymbol{\sigma}_\mathbf{f}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t})] : \boldsymbol{\sigma} \quad \text{(I-Lambda)}$$

$$\frac{\mathbf{N} \Vdash \mathbf{\Sigma} \xrightarrow{\mathsf{n_i}} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ [\cdot]) : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \quad \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ [\cdot]) : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \xrightarrow{@\mathsf{v}} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ \mathbf{v}) : \boldsymbol{\sigma}_2}{\mathbf{N} \Vdash \mathbf{\Sigma} \xrightarrow{\gg \mathsf{n_i}\ ;\ \mathsf{v}} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet (\mathbf{t}\ \mathbf{v}) : \boldsymbol{\sigma}_2} \quad \text{(Call-N)}$$

$$\frac{\mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[({}^{\boldsymbol{\sigma_1} \to \boldsymbol{\sigma_2}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t})\ \mathbf{v})] : \boldsymbol{\sigma} \xrightarrow{(\lambda \mathsf{x.t})} \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma} \bullet \mathbf{v} : \boldsymbol{\sigma}_1 \quad \mathbf{N} \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma} \bullet \mathbf{v} : \boldsymbol{\sigma}_1 \xrightarrow{\mathsf{v}} \mathbf{N}' \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma}}{\mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{E}[({}^{\boldsymbol{\sigma_1} \to \boldsymbol{\sigma_2}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t})\ \mathbf{v})] : \boldsymbol{\sigma} \xrightarrow{\gg (\lambda \mathsf{x.t})\ ;\ \mathsf{v}} \mathbf{N}' \Vdash \mathbf{\Sigma}, \mathbf{E} : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma}} \quad \text{(Call-L)}$$

$$\frac{\overline{\mathsf{C}} \mid\mid \mathbf{S_e} \twoheadrightarrow \overline{\mathsf{C}} \mid\mid \mathbf{S_e}'}{\mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t} : \boldsymbol{\sigma} \xrightarrow{\tau^+} \mathbf{N} \Vdash \mathbf{\Sigma} \bullet \mathbf{t}' : \boldsymbol{\sigma}} \quad \text{(Silent)}$$

Figure 4: The Labelled Transition System over a λ_s-calculus configuration $\mathbf{S}$.

λ_a-configuration can insert into a λ_s-configuration. The labelled reduction rules Call-L and Call-N describe the values the λ_a-configuration observes when either configuration calls a function. The labelled reduction rules I-Wrong and O-Done denote failure and termination. Reduction steps that only modify the λ_s-configuration cannot be observed by a λ_a-configuration and are thus labelled as silent, as denoted in Silent.

Like for the bisimulation over the λ_s-calculus we define a weak bisimulation over this LTS. Define the transition relation $\mathtt{t} \overset{\gamma^+}{\Longrightarrow} \mathtt{t}'$ as $\mathtt{t} \xrightarrow{\tau^+}{}^{*} \xrightarrow{\gamma^+} \mathtt{t}'$ where $\xrightarrow{\tau^+}{}^{*}$ is the reflexive transitive closure of the silent transitions $\xrightarrow{\tau^+}$.

Bisimulation is now defined as follows:

Definition 4. *The relation $\mathcal{S}_+$ is a **bisimulation** if and only if $\mathbf{S}_1\ \mathcal{S}_+\ \mathbf{S}_2$ implies:*

(1) Given $\mathbf{S}_1 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_1'$ there is $\mathbf{S}_2' : \mathbf{S}_2 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_2' \wedge \mathbf{S}_1'\ \mathcal{S}_+\ \mathbf{S}_2'$

(2) Given $\mathbf{S}_2 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$ there is $\mathbf{S}_1' : \mathbf{S}_1 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_1' \wedge \mathbf{S}_1'\ \mathcal{S}_+\ \mathbf{S}_2'$

We denote bisimilarity, the largest bisimulation, as $\approx_+$.

5.2.3 Proof of Congruence

The proof that bisimilarity corresponds to contextual equivalence is split into three sublemma: preservation, completeness and soundness.

Lemma 1. (Preservation)
If $\mathbf{S}_1 \simeq_c \mathbf{S}_2$ and $\mathbf{S}_1 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_1'$ and $\mathbf{S}_2 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$ then $\mathbf{S}_1' \simeq_c \mathbf{S}_2'$

To simplify the proof of the completeness lemma, we show that the LTS transitions preserve contextual equivalence.
Proof Sketch. We assume that:

$$\mathbf{S}_1 \simeq_c \mathbf{S}_2 \wedge \mathbf{S}_1 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_1' \wedge \mathbf{S}_2 \overset{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$$

We must show that:

$$\mathbf{S}_1' \simeq_c \mathbf{S}_2'$$

The proof proceeds by case analysis on the label γ^+. In this proof sketch we consider two broad cases: the labels:

$$(\lambda \mathsf{x.t}),\ \mathsf{n},\ \mathsf{true},\ \mathsf{false},\ \mathsf{wrong},\ \mathsf{done},\ \gg (\lambda \mathsf{x.t})\ ;\ \mathsf{v}$$

describe the terms that the λ_a-context receives from either configuration. Because identical labels cannot be distinguished by a λ_a-context the thesis holds. The labels:

$$@\mathsf{v},\ \gg \mathsf{n}\ ;\ \mathsf{v}$$

describe the modifications to the configuration $\mathbf{S}_1$ and $\mathbf{S}_2$. We show that the thesis holds by *reductio ad absurdum*. For the thesis not to hold there must be an input label that results in a modification to the configurations $\mathbf{S}_1$ and $\mathbf{S}_2$ that cannot be performed by a λ_a-context A. This is not the case for either label. □

Lemma 2. (Completeness) $\mathbf{S}_1 \simeq_c \mathbf{S}_2 \Rightarrow \mathbf{S}_1 \approx_+ \mathbf{S}_2$

A proof sketch for this lemma that relies on Lemma 1 is available in Appendix A.2.

As by Gordon [10] we show that contextual equivalence is itself a bisimulation relation, by case analysis on the LTS labels γ^+.

Lemma 3. (Soundness) $\mathbf{S}_1 \approx_+ \mathbf{S}_2 \Rightarrow \mathbf{S}_1 \simeq_c \mathbf{S}_2$

A full proof of this lemma is available in Appendix A.3. The proof follows by induction on the number of reduction steps. We show that given $P_1 = \mathsf{A} \mid\mid \mathbf{S}_1$ and $P_2 = \mathsf{A} \mid\mid \mathbf{S}_2$ that P_2 diverges if P_1 diverges.

We now conclude that bisimilarity $\approx_+$ coincides with contextual equivalence $\simeq_c$.

Theorem 4 (Full Abstraction of the Bisimilarity)**.**
$\mathbf{S}_1 \simeq_c \mathbf{S}_2 \;\Leftrightarrow\; \mathbf{S}_1 \approx_+ \mathbf{S}_2$

Proof. The theorem follows from Lemma 2 and Lemma 3. □

5.3 A Fully Abstract Compilation Scheme

Securely compiling a λ_s-calculus term $\mathbf{t}$ is done by producing the following λ^+-configuration $\mathbf{S}$:

$$\mathbf{t}^{\downarrow} = \star \Vdash \boldsymbol{\varepsilon} \bullet \mathbf{t} : \boldsymbol{\sigma}$$

where $\boldsymbol{\Gamma} \vdash \mathbf{t} : \boldsymbol{\sigma}$.

We must now prove that this compilation scheme preserves and reflects the equivalences of the λ_s-calculus. We do so by showing that bisimilar terms in the λ_s-calculus coincide with bisimilar compiled terms in the λ^+-calculus. The proof is divided into three sublemma:

Lemma 4. (Compiler Correctness)
$\mathbf{t_1} \rightarrow^* \mathbf{v} \iff \mathbf{N} \Vdash \boldsymbol{\Sigma} \bullet \mathbf{t} \twoheadrightarrow^* \mathbf{N} \Vdash \boldsymbol{\Sigma} \bullet \mathbf{v}$

Proof Sketch. This follows from the fact that the semantics of the λ_s-calculus are preserved in the λ^+-calculus. □

Lemma 5. (Comp. Preservation) $\mathbf{t_1} \approx_s \mathbf{t_2} \Rightarrow \mathbf{t_1}^{\downarrow} \approx_+ \mathbf{t_2}^{\downarrow}$

A proof sketch is available in Appendix A.4. The proof sketch proceeds by coinduction and case analysis on the label γ^+.

Lemma 6. (Comp. Reflection) $\mathbf{t_1}^{\downarrow} \approx_+ \mathbf{t_2}^{\downarrow} \Rightarrow \mathbf{t_1} \approx_s \mathbf{t_2}$

A proof sketch is available in Appendix A.5. We prove the lemma by the contrapositive: $\mathbf{t_1} \not\approx_s \mathbf{t_2} \Rightarrow \mathbf{t_1}^{\downarrow} \not\approx_+ \mathbf{t_2}^{\downarrow}$. The proof proceeds by induction because if we have that $\mathbf{t_1} \not\approx_s \mathbf{t_2}$ then there is a finite sequence of LTS transitions until the bisimilarity $\approx_s$ fails to hold. We show that any such sequence can be reproduced in the λ^+-calculus bisimulation.

We conclude that the compilation scheme is fully abstract.

Theorem 5 (Full Abstraction of the Compilation Scheme)**.**
$\mathbf{t_1} \approx_s \mathbf{t_2} \iff \mathbf{t_1}^{\downarrow} \approx_+ \mathbf{t_2}^{\downarrow}$

Proof. The theorem follows from Lemma 5 and Lemma 6. □

6. Related Work

Language Interoperation Techniques that ensure secure interoperation between languages with different levels of abstraction, have been developed before. Furr and Foster address the complications that arise when OCaml interoperates with C, by developing a multi-language type system that embeds OCaml types in C and vice-versa [8]. They however do not consider the fact that a C program can be an attacker capable circumventing their typing system by directly accessing the OCaml memory structures.

Tan et al. tackle the issues that arise when Java interoperates with C through SafeJNI [23], a framework that ensures type safety through the Java foreign function interface. Their system however, requires both static and dynamic checks on the C code that Java interoperates with. Our technique for secure operational semantics, on the other hand, does not require any static checks on the attacker.

Gampe et. al present a technique that establishes the noninterference properties of two interoperating languages with different security typing mechanisms [9]. They do not consider any attacker model.

Matthews' and Findler's multi-language semantics [15] provide a technique for specifying operational semantics that allows two languages to interoperate in a way that preserves termination and type safety. In their work however, they aim to abstract away low-level details and instead focus on semantic properties. Our technique in contrast, focusses on lifting low-level properties into the operational semantics.

Zdancewic et al. present a multi-agent calculus that treats the different modules that make up a program as different principals, each with a different view of the environment [26]. Their work however models the different views each agents sees through typing.

Fully Abstract Compilation Secure (fully abstract) compilation was first introduced by Abadi [1] as a criticism of the way Java was translated into the Java bytecode language, and of the way π-calculus was translated into the spi-calculus. Secure compilation has since been applied on many different source languages, such as the λ-calculus [2], the λ-calculus extended with dynamic memory allocation [12] and JavaScript [7].

This paper further developed on the secure compilation schemes of Agten et. al [4] and Patrignani et al. [19], which extend the target language of their secure compilers with a memory protection mechanism.

Bisimulation Bisimulation has been applied to the λ-calculus before, most notably by Abramsky in his work on an applicative bisimulation for the lazy λ-calculus [3].

In this work we relied on Gordon's proofs, LTS and bisimulation statement for FPC [10]. Gordon's approach leverages Howe's proof method for bisimulation [11], whose syntactical approach greatly simplifies Abramsky's domain logic based technique.

Both Sumii and Pierce and Jeffrey and Rathke have defined bisimulations for a λ-calculus with name generation [13, 22]. Our definition of bisimulation is however much simpler than their respective definitions, as the names in our multi-language system are both global and enumerable.

7. Conclusions & Future Work

This paper introduced operational semantics that preserve the abstraction of a simply typed λ-calculus that interoperates with the λ-calculus model of an arbitrary machine-level attacker. These operational semantics lift the low-level memory protection techniques from the PMA mechanism into the resulting multi-language system.

There are several directions for future work. One is to investigate a more fine-grained approach to secure interoperation. One could imagine, for example, a multi-language system where one must consider different types of attackers. In such a scenario the goal should be to adapt the operational semantics between two interoperating components to the level of trust between both components. If a components attacker model is not capable of intercepting function calls, for example, the interoperation with that component should be more symmetrical.

Another possibility is to research secure interoperation semantics for concurrent languages. Given that our technique already removes the direct term embedding found in most existing multi-language techniques it might be interesting to remove the execution order dependencies as well.

Acknowledgments This work has been supported in part by the Intel Labs University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme 840 of the European Union (B-CCENTRE).

References

[1] M. Abadi. Protection in programming-language translations. In *Secure Internet Programming*, volume 1603 of *LNCS*, 1999.

[2] M. Abadi and G. Plotkin. On protection by layout randomization. In *CSF '10*. IEEE, 2010.

[3] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[4] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*, CSF 2012. IEEE, 2012.

[5] P.-L. Curien. Definability and full abstraction. *Electronic Notes on Theoretical Computer Science*, 172, Apr. 2007.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.

[7] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL '13*, New York, NY, USA, 2013. ACM.

[8] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, New York, NY, USA, 2005. ACM.

[9] A. Gampe and J. von Ronne. Security completeness: Towards noninterference in composed languages. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[10] A. Gordon. *Bisimilarity as a Theory of Functional Programming: Mini-course*. BRICS notes series. Computer Science Department, 1995.

[11] D. J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989)*, pages 198–203. IEEE Computer Society Press, June 1989.

[12] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF '11*. IEEE, 2011.

[13] A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. Computer Science Report 02-2000, University of Sussex, 2000.

[14] A. Jeffrey and J. Rathke. Java JR: fully abstract trace semantics for a core Java language. In *Proceedings of the 14th European conference on Programming Languages and Systems*, ESOP'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[15] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3), Apr. 2009.

[16] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*. ACM, 2013.

[17] M. Milenković, A. Milenković, and E. Jovanov. Using instruction block signatures to counter code injection attacks. *SIGARCH Comput. Archit. News*, 33(1):108–117, Mar. 2005.

[18] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22nd USENIX conference on Security symposium*. USENIX Association, 2013.

[19] M. Patrignani, D. Clarke, and F. Piessens. Secure compilation of object-oriented components to protected module architectures. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2013.

[20] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[21] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In S. Jajodia and J. Zhou, editors, *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering: Security and Privacy in Communication Networks,*. Springer, Sept. 2010.

[22] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 161–172, New York, NY, USA, 2004. ACM.

[23] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. Safe Java native interface. In *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006.

[24] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[26] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: a syntactic proof technique. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, New York, NY, USA, 1999. ACM.

A. Proof Sketches and Proofs

A.1 The Bisimulation $\approx_s$ is a congruence

The definition of contextual equivalence is extended with a definition of contextual order $\cong_s$.

Definition 5. Contextual equivalence ($\simeq_s$) *and* **Contextual Order** ($\cong_s$) *are defined as:*

(1) $\mathbf{t_1} \cong_s \mathbf{t_2} \iff \forall \mathbf{C}.\, \exists \mathbf{b}.\, \mathbf{C[t_1]} \rightarrow^* \mathbf{b} \Rightarrow \mathbf{C[t_2]} \rightarrow^* \mathbf{b}$

(2) $\mathbf{t_1} \simeq_s \mathbf{t_2} \iff \mathbf{t_1} \cong_s \mathbf{t_2}\ \&\ \mathbf{t_2} \cong_s \mathbf{t_1}$

A non-symmetrical definition of bisimilarity, similarity is introduced as well.

Definition 6. Simulation *The relation $\mathcal{S}_s$ is defined as:*

(1) Given $\mathbf{t_1} \stackrel{\gamma}{\Rightarrow} \mathbf{t_1}'$ *there is* $\mathbf{t_2}' : \mathbf{t_2} \stackrel{\gamma}{\Rightarrow} \mathbf{t_2}' \wedge \mathbf{t_1}'\ \mathcal{S}_s\ \mathbf{t_2}'$

We denote similarity, as the largest simulation $\lesssim_s$. Relying on the symmetry property of both bisimilarity and contextual equivalence, the proof is simplified to one that establishes that similarity equals contextual order. The proof is split into two sublemma:

Lemma 7. (Completeness) $\mathbf{t_1} \cong_s \mathbf{t_2} \Rightarrow \mathbf{t_1} \lesssim_s \mathbf{t_2}$

Proof sketch. To prove that contextual order implies similarity it is establish that contextual order is itself a similarity. Assume that :

$$\mathbf{C[t_1]} \cong_s \mathbf{C[t_2]}\ and\ \mathbf{t_1} \stackrel{\gamma}{\Rightarrow} \mathbf{t_1}'$$

We want to show that:

$$\mathbf{t_2} \stackrel{\gamma}{\Rightarrow} \mathbf{t_2}' and\ \mathbf{t_1}' \cong_s \mathbf{t_2}'$$

We restrict this proof sketch to the most interesting case: the label $@\mathbf{v}$.

From the contextual equivalence between $\mathbf{t_1}$ and $\mathbf{t_2}$ it follows that $\mathbf{t_1}$ and $\mathbf{t_2}$ are both function types and thus both reduce to λ-terms $\mathbf{v_1}$ and $\mathbf{v_2}$. Given that $\boldsymbol{\gamma} = @\mathbf{v}$ it now follows from the LTS that $\mathbf{t_1'} = (\mathbf{v_1}\ \mathbf{v})$ and $\mathbf{t_2'} = (\mathbf{v_2}\ \mathbf{v})$. The label $@\mathbf{v}$ can thus be encoded as the context $\mathbf{C} = ([\cdot]\ v)$, because contextual order is closed under contexts we can now conclude that: $(\mathbf{v_1}\ \mathbf{v}) \cong_\mathbf{s} (\mathbf{v_2}\ \mathbf{v})$. □

Proving the reverse, that similarity implies contextual order, is done by applying Howe's method [11]. Doing so requires the construction of a similarity candidate $\lesssim_b$ that includes its own compatible refinement $\widehat{\lesssim_\bullet}$. The compatible refinement $\widehat{\lesssim_\bullet}$ relates the following well-typed λ_s-terms: equal constants, variables and outer forms that are related by $\lesssim_\bullet$. It relates λ-terms, for example, as follows:

$$\frac{\boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\sigma} \vdash \mathbf{t_1}\ \lesssim_\bullet\ \mathbf{t_1'}}{\boldsymbol{\Gamma} \vdash (\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t_1})\ \widehat{\lesssim_\bullet}\ (\lambda \mathbf{x} : \boldsymbol{\sigma}.\mathbf{t_1'})}$$

in a sense this compatible refinement relation constructs all possible contexts a λ_s-term may be placed in.
The similarity candidate $\lesssim_b$ is defined as:

$$\frac{\boldsymbol{\Gamma} \vdash \mathbf{t}\ \widehat{\lesssim_\bullet}\ \mathbf{t'} \qquad \boldsymbol{\Gamma} \vdash \mathbf{t'} \lesssim_s\ \mathbf{t''}}{\boldsymbol{\Gamma} \vdash \mathbf{t}\ \lesssim_\bullet\ \mathbf{t''}}$$

Lemma 8. (Soundness) $\mathbf{t_1} \lesssim_s \mathbf{t_2} \Rightarrow \mathbf{t_1} \cong_s \mathbf{t_2}$

Proof sketch. We first prove that the similarity candidate $\lesssim_\bullet$ is an alternative definition of $\lesssim_s$: $\lesssim_\bullet = \lesssim_b$.

1. $\lesssim_s \subseteq \lesssim_\bullet$. Given that $\lesssim_s$ is reflexive, it follows from structural induction on t that: $\lesssim_\bullet$ is reflexive and $\widehat{\lesssim_\bullet}$ is reflexive. Applying the fact that $\widehat{\lesssim_\bullet}$ is reflexive to the definition of $\widehat{\lesssim_\bullet}$ allows us to conclude that: $\lesssim_s \subseteq \lesssim_\bullet$.

2. $\lesssim_\bullet \subseteq \lesssim_s$. We show that $\lesssim_\bullet$ is a simulation, by rule induction on $\mathbf{t} \stackrel{\gamma}{\mapsto} \mathbf{t'}$. The inductive case is: $\mathbf{t_1} \xrightarrow{\tau} \mathbf{t_1''} \xrightarrow{\gamma} \mathbf{t_1'}$. We must show that $\mathbf{t_1''} \lesssim_\bullet \mathbf{t_2}$. This will allow us to conclude by the induction hypothesis that $\mathbf{t_2} \xrightarrow{\gamma} \mathbf{t_2'}$ and that $\mathbf{t_1'} \lesssim_\bullet \mathbf{t_2'}$. To do so we must examine each internal reduction step.

   The λ_s-calculus has two internal reduction steps: function application and if reduction. Both cases are covered in the Gordon proof [10]. □

Now assume that:

$$\mathbf{C[t_1]} \rightarrow^* \mathbf{b}\ and\ \mathbf{t_1} \lesssim_s \mathbf{t_2}\ and\ \mathbf{C[t_1]} \lesssim_s \mathbf{C[t_2]}$$

There are two cases:

- $\mathbf{b} = \mathbf{true}$. From the LTS it follows that:

  $$\mathbf{C[t_1]} \stackrel{\mathbf{true}}{\longmapsto} \#\mathbf{t}$$

  From the definition of $\lesssim_s$ we conclude that:
  $\mathbf{C[t_2]} \stackrel{\mathbf{true}}{\longmapsto} \#\mathbf{t}$ and thus that $\mathbf{C[t_2]} \rightarrow^* \#\mathbf{t}$.
- $\mathbf{b} = \mathbf{false}$, *mutatis mutandis*. □

We now conclude that bisimilarity is a full abstraction of contextual equivalence.

Proof. Proof of Theorem 3:

$$\mathbf{t_1} \simeq_s \mathbf{t_2} \Leftrightarrow \mathbf{t_1} \approx_s \mathbf{t_2}$$

The theorem follows from Lemma 7 and 8 and the symmetrical properties of bisimilarity and contextual equivalence. □

A.2 Completeness

Proof Sketch. Proof sketch of Lemma 2:

$$\mathbf{S}_1 \simeq_c \mathbf{S}_2 \Rightarrow \mathbf{S}_1 \approx_+ \mathbf{S}_2$$

Proving that contextual equivalence implies bisimilarity is done by showing that the contextual equivalence relation is itself a bisimulation.
Assume that: $\mathbf{S}_1 \simeq_c \mathbf{S}_2$.
Because bisimilarity is symmetrical, we divide the proof into two parts:

1. Assume that: $\mathbf{S}_1 \stackrel{\gamma^+}{\Longrightarrow} \mathbf{S}_1'$.
   We now want to show that there exists a $\mathbf{S}_2'$ such that:

   (a) $\mathbf{S}_2 \stackrel{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$

   (b) $\mathbf{S}_1' \simeq_c \mathbf{S}_2'$

   The second thesis follows immediately from Lemma 1. We prove the first thesis by case analysis on the label γ^+. For every label γ^+ we establish the thesis by *reductio ad absurdum*: For reductio assume that $\nexists \mathbf{S}_2'.\mathbf{S}_2 \stackrel{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$ then $\mathbf{S}_1 \not\simeq_c \mathbf{S}_2$. Note that we don't consider the case: $\nexists \gamma^+.\mathbf{S}_2 \stackrel{\gamma^+}{\Longrightarrow} \mathbf{S}_2'$ as this implies that the configuration $\mathbf{S}_2$ diverges or gets stuck which by Theorem 1 is not possible. Also note that we further simplify the proof by only establishing that there exists a context that can distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$, to comply with the definition of contextual equivalence the context must also diverge in one case. In this proof sketch we restrict ourselves to the most informative cases:

   - $\gamma^+ = \mathsf{n_i}$:: By the reduction rule S-Name we have that the λ_a-context receives a name $\mathsf{n_i}$ from the configuration $\mathbf{S}_1$. A simple λ_a-context $\mathsf{A} = [\cdot]$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ does not produce the same observable label $\mathsf{n_i}$.
   - $\gamma^+ = (\lambda \mathsf{x.t})$:: By the reduction rule S-Call we have that the λ_a-context recieves a λ-term $(\lambda \mathsf{x.t})$ form the configuration $\mathbf{S}_1$. A simple λ_a-context $\mathsf{A} = (\lambda \mathsf{x.t}) \equiv \cdot$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ does not produce an observable label $(\lambda \mathsf{x.t})$.
   - $\gamma^+ = @\mathsf{n_i}$:: By the reduction rule A-Name we have that the λ_a-context can only apply the label $@\mathsf{n_i}$ if the configuration is passive and the name is of the correct type. A simple λ_a-context $\mathsf{A} = \mathsf{n_i}$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ cannot produce the observable label $@\mathsf{n_i}$.
   - $\gamma^+ = \gg (\lambda \mathsf{x.t}) \ ; \ \mathsf{v}$:: By the reduction rules we have that the λ_a-context will observe $((\lambda \mathsf{x.t}) \ \mathsf{v})$ from the first configuration $\mathbf{S}_1$. The context observes the application as a hole as by the LTS and the reductions rules we have that when the attacker resumes control it will either perceives the full term or wrong if an incorrect value was applied. In the latter case a different label wrong is observed from $\mathbf{S}_1$. A λ_a-context $\mathsf{A} = (((\lambda \mathsf{x.t}) \ \mathsf{v}) \equiv [\cdot]$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ does not produce an observable label $\gg (\mathsf{lamx.t}) \ ; \ \mathsf{v}$.
   - $\gamma^+ = \gg \mathsf{n_i} \ ; \ \mathsf{v}$:: By the reduction rule A-Call we have that the λ_a-context can apply the label $\gg \mathsf{n_i} \ \mathsf{v}$ if the configuration is passive and the name is in the domain. By the reduction rules WrongN, WrongC, WrongL and WrongB we have that v must be of the correct type. Like the $\gamma^+ = @\mathsf{n_i}$ case a λ_a-context $\mathsf{A} = (\mathsf{n_i} \ \mathsf{v})$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ cannot produce the observable label $\gg \mathsf{n_i} \ ; \ \mathsf{v}$.
   - $\gamma^+ = \mathsf{wrong}$:: By the reduction rules WrongN, WrongC, WrongL and WrongB we have that the $\mathbf{S}_1$ produces a label wrong if the λ_a-context passes an incorrectly typed value to $\mathbf{S}_1$. For any value v that is incorrectly typed for the configuration $\mathbf{S}_1$ a λ_a-context $\mathsf{A} = \mathsf{v}$ can thus distinguish between $\mathbf{S}_1$ and $\mathbf{S}_2$ if $\mathbf{S}_2$ does not produce the observable label wrong for the same context A.

2. As in case 1, *mutatis mutandis*. □

A.3 Soundness Proof

Proof. Proof of Lemma 3.

$$\mathbf{S}_1 \approx^+ \mathbf{S}_2 \Rightarrow \mathbf{S}_1 \simeq_c \mathbf{S}_2$$

As mentioned in Section 5.2.1 the thesis $\mathbf{S}_1 \simeq_c \mathbf{S}_2$ becomes $\forall \mathsf{A}.\ \mathsf{A} \mid\mid \mathbf{S}_1 \Uparrow \iff \mathsf{A} \mid\mid \mathbf{S}_2 \Uparrow$ The proof is divided into two cases, one case for each side of the co-implication.

1. $\Rightarrow$ In this case the thesis is $\forall \mathsf{A}.\ \mathsf{A} \mid\mid \mathbf{S}_1 \Uparrow \Rightarrow \mathsf{A} \mid\mid \mathbf{S}_2 \Uparrow$. The thesis can be redefined as:

$$\forall \mathsf{A}.\ \forall k \in \mathbb{N}.\ \mathsf{A} \mid\mid \mathbf{S}_1 \twoheadrightarrow^k \mathsf{A}'_1 \mid\mid \mathbf{S}'_1 \Rightarrow \\ \forall m \in \mathbb{N}.\ \mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^m \mathsf{A}'_2 \mid\mid \mathbf{S}'_2$$

   The proof proceeds by induction on m.

   Base case: $m = 0$**.** Straightforward: $\mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^0 \mathsf{A} \mid\mid \mathbf{S}_2$.

   Inductive case: $m = h + 1$**.** The thesis is:

$$\mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^{h+1} \mathsf{A}'_2 \mid\mid \mathbf{S}'_2.$$

   The inductive hypotheses (IH) is:

$$\forall \mathsf{A}.\ \forall k \in \mathbb{N}.\ \mathsf{A} \mid\mid \mathbf{S}_1 \twoheadrightarrow^k \mathsf{A}'_1 \mid\mid \mathbf{S}'_1 \Rightarrow \\ \mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^h \mathsf{A}^h_2 \mid\mid \mathbf{S}^h_2$$

   We know from this IH that:

$$\exists \mathsf{A}.\mathbf{S}_1.\ \mathsf{A} \mid\mid \mathbf{S}_1 \twoheadrightarrow^h \mathsf{A}^h_1 \mid\mid \mathbf{S}^h_1 \twoheadrightarrow^{k-h} \mathsf{A}'_1 \mid\mid \mathbf{S}'_1$$

   We prove the thesis by reasoning about what the presence or absence of the last observable label γ^+ tells us about the existence of a next reduction step $h + 1$. There are two cases: either the λ_a-context A is passive or executing.

   (a) The λ_a-context is passive: $\mathsf{A} = \overline{\mathsf{C}}$ and the λ_s-configuration is executing: $\mathbf{S} = \mathbf{S_e}$.
   In this case there are two sub-cases:

      i. $\exists \gamma^+.\ \mathbf{S_e}^h_1 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_1$.
      By the assumption $\mathbf{S_{e}}_1 \approx^+ \mathbf{S_{e}}_2$ we conclude that $\mathbf{S_e}^h_2 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_2$ and $\mathbf{S}^{h'}_1 \approx^+ \mathbf{S}^{h'}_2$. This, in conjunction with the IH, implies the thesis:

$$\mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^{h+1} \mathsf{A}' \mid\mid \mathbf{S}'_2.$$

      ii. $\nexists \gamma^+.\ \mathbf{S_e}^h_1 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_1$.
      An executing λ_s-configuration $\mathbf{S_e}$ does not produce an observable label if and only if it is diverging. However because λ_s-configurations are stacks of well typed λ_s-terms this is not possible.

   (b) The λ_a-context is executing: $\mathsf{A} = \mathsf{t}$ and the λ_s-configuration is passive $\mathbf{S} = \mathbf{S_p}$.
   In this case there are two sub-cases as well:

      i. $\exists \gamma^+.\ \mathbf{S_p}^h_1 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_1$.
      Because the observable label γ^+ is produced by the respective λ_a-configurations, we must thus show that: $\mathsf{A}^h_1 = \mathsf{A}^h_2$, where the existence of A^h_2 derives from the induction hypothesis.

      We know by the assumption: $\mathbf{S}_1 \approx^+ \mathbf{S}_2$ that both λ_a-configurations were modified by the same stream of observable labels if there are any such labels : $\exists k \in \mathbb{N}.k \leq h \wedge \mathsf{A} \mid\mid \mathbf{S}_1 \twoheadrightarrow^k \mathsf{A}^k_1 \mid\mid \mathbf{S}^k_1 \wedge \mathsf{A}^k_1 \mid\mid \mathbf{S}^k_1 \twoheadrightarrow^{h-k} \mathsf{A}^h_1 \mid\mid \mathbf{S}^h_1$ where $\mathbf{S}_1 \xRightarrow{\overline{\gamma^+}} \mathbf{S}^k_1$ and that $\exists k \in \mathbb{N}.k \leq h \wedge \mathsf{A} \mid\mid \mathbf{S}_2 \twoheadrightarrow^k \mathsf{A}^k_2 \mid\mid \mathbf{S}^k_2 \wedge \mathsf{A}^k_2 \mid\mid \mathbf{S}^k_2 \twoheadrightarrow^{h-k} \mathsf{A}^h_2 \mid\mid \mathbf{S}^h_2$ where $\mathbf{S}_2 \xRightarrow{\overline{\gamma^+}} \mathbf{S}^k_2$.
      Combining the fact that the reduction rules of the λ^+-calculus are deterministic and with the fact that the λ_a-contexts are updated in the same way by identical labels γ^+ we conclude that $\mathsf{A}^h_1 = \mathsf{A}^h_2$ and that $\mathbf{S_p}^h_2 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_2$. This implies the thesis.

      ii. $\nexists \gamma^+.\ \mathbf{S_p}^h_1 \xRightarrow{\gamma^+} \mathbf{S}^{h'}_1$.
      If there exists no label γ^+ the λ_a-context is diverging. In the previous case we established that $\mathsf{A}^h_1 = \mathsf{A}^h_2$. As such both $\mathsf{A} \mid\mid \mathbf{S}_1$ and $\mathsf{A} \mid\mid \mathbf{S}_2$ divergence, which implies the thesis.

2. $\Leftarrow$ As in case 1, *mutatis mutandis*.

□

A.4 Compiler Preservation

Proof Sketch. Proof sketch of Lemma 5:

$$\mathbf{t_1} \approx_s \mathbf{t_2} \Rightarrow \mathbf{t_1}^{\downarrow} \approx_+ \mathbf{t_2}^{\downarrow}$$

We must develop a relation $\mathcal{R}$ such that:

$$\mathbf{t_1}^{\downarrow} \mathcal{R} \mathbf{t_2}^{\downarrow} \tag{1}$$

and that for all $\mathbf{S_1} \mathcal{R} \mathbf{S_2}$ we have that:

$$\mathbf{S_1} \xRightarrow{\gamma^+} \mathbf{S_1}' \wedge \exists \mathbf{S_2}'.\mathbf{S_2} \xRightarrow{\gamma^+} \mathbf{S_2}' \Rightarrow \mathbf{S_1}' \mathcal{R} \mathbf{S_2}' \tag{2}$$

$$\mathbf{S_2} \xRightarrow{\gamma^+} \mathbf{S_2}' \wedge \exists \mathbf{S_1}'.\mathbf{S_1} \xRightarrow{\gamma^+} \mathbf{S_1}' \Rightarrow \mathbf{S_1}' \mathcal{R} \mathbf{S_2}' \tag{3}$$

We build the following relation $\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1$:

$$\mathcal{R}_0 = \{(\mathbf{N_1} \Vdash \mathbf{\Sigma_1}, \mathbf{N_2} \Vdash \mathbf{\Sigma_2}) \mid \forall \mathbf{N_1}, \mathbf{N_2}, \mathbf{\Sigma_1}, \mathbf{\Sigma_2} \textit{ such that } \mathbf{N_1} \approx_N \mathbf{N_2} \textit{ and } \mathbf{\Sigma_1} \approx_{\mathbf{\Sigma}} \mathbf{\Sigma_2}\}$$

$$\mathcal{R}_1 = \{(\mathbf{N_1} \Vdash \mathbf{\Sigma_1} \bullet \mathbf{t_1}, \mathbf{N_2} \Vdash \mathbf{\Sigma_2} \bullet \mathbf{t_2}) \mid \forall \mathbf{N_1}, \mathbf{N_2}, \mathbf{\Sigma_1}, \mathbf{\Sigma_2}, \mathbf{t_1}, \mathbf{t_2} \textit{ such that } (\mathbf{N_1} \Vdash \mathbf{\Sigma_1}, \mathbf{N_2} \Vdash \mathbf{\Sigma_2}) \in \mathcal{R}_0 \textit{ and } \mathbf{t_1} \approx_s \mathbf{t_2}\}$$

where $\approx_N$ is defined as:

$$\mathbf{N_1} \approx_N \mathbf{N_2} \iff dom(\mathbf{N_1}) = dom(\mathbf{N_2}) \textit{ and } \forall \mathsf{n_i} \in dom(\mathbf{N_1}).\, \mathbf{N_1}(\mathsf{n_i}) \approx_s \mathbf{N_2}(\mathsf{n_i})$$

and where $\approx_{\mathbf{\Sigma}}$ is defined as:

$$\mathbf{\Sigma_1} \approx_{\mathbf{\Sigma}} \mathbf{\Sigma_2} \iff \textit{For } (\mathbf{E_1^1}, ..., \mathbf{E_n^1}) : (\boldsymbol{\sigma_1^1}, ..., \boldsymbol{\sigma_n^1}) \textit{ in } \mathbf{\Sigma_1} \textit{ and } (\mathbf{E_1^2}, ..., \mathbf{E_{n'}^2}) : (\boldsymbol{\sigma_1^2}, ..., \boldsymbol{\sigma_n^2}) \textit{ in } \mathbf{\Sigma_2}$$
$$\textit{we have } \mathbf{n} = \mathbf{n}' \textit{ and for every } \mathbf{1} \leq \mathbf{i} \leq \mathbf{n} : \boldsymbol{\sigma_i^1} = \boldsymbol{\sigma_i^2} \textit{ and } \forall \mathbf{t}', \mathbf{t}''.\mathbf{\Gamma} \vdash \mathbf{t}' : \boldsymbol{\sigma_i^1} \wedge \mathbf{\Gamma} \vdash \mathbf{t}'' : \boldsymbol{\sigma_i^1} \wedge \mathbf{t}' \approx_s \mathbf{t}'' \Rightarrow \mathbf{E_i^1}[\mathbf{t}'] \approx_{\mathbf{s}} \mathbf{E_i^2}[\mathbf{t}'']$$

We now proof the three cases.

- In case (1) we have that $\star \Vdash \boldsymbol{\varepsilon} \bullet \mathbf{t_1} \mathcal{R} \star \Vdash \boldsymbol{\varepsilon} \bullet \mathbf{t_2}$ as we have that $\mathbf{t_1} \approx_s \mathbf{t_2}$ from the assumption.
- In case (2) we proceed by case analysis on γ^+. We restrict this proof sketch to the most interesting labels.
  - $\gamma^+ = \mathsf{n_i}$: By the LTS we have that:

$$\mathbf{S_1} = \mathbf{N_1} \Vdash \mathbf{\Sigma_1} \bullet (\lambda \mathbf{x}.\mathbf{t_1}) : \boldsymbol{\sigma} \overset{\mathsf{n_i}}{\Rightarrow} \mathbf{N_1}[\mathsf{n_i} \mapsto ((\lambda \mathbf{x}.\mathbf{t_1}), \boldsymbol{\sigma})] \Vdash \mathbf{\Sigma_1} = \mathbf{S_1}'$$

    and that:

$$\mathbf{S_2} = \mathbf{N_2} \Vdash \mathbf{\Sigma_2} \bullet (\lambda \mathbf{x}.\mathbf{t_2}) : \boldsymbol{\sigma} \overset{\mathsf{n_i}}{\Rightarrow} \mathbf{N_2}[\mathsf{n_i} \mapsto ((\lambda \mathbf{x}.\mathbf{t_2}), \boldsymbol{\sigma})] \Vdash \mathbf{\Sigma_2} = \mathbf{S_2}'$$

    Only the maps are modified in this case. By $\mathbf{S_1} \mathcal{R} \mathbf{S_2}$ we have that $(\lambda \mathbf{x}.\mathbf{t_1}) \approx_{\mathbf{s}} (\lambda \mathbf{x}.\mathbf{t_2})$. We conclude that the thesis: $\mathbf{S}_1' \mathcal{R} \mathbf{S}_2'$ holds.
  - $\gamma^+ = @\mathsf{n_i}$: By the LTS we have that:

$$\mathbf{S_1} = \mathbf{N_1}[\mathsf{n_i} \mapsto (\lambda \mathbf{x}.\mathbf{t_1})] \Vdash \mathbf{\Sigma_1}, \mathbf{E} : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \overset{@\mathsf{n_i}}{\Longrightarrow} \mathbf{N_1} \Vdash \mathbf{\Sigma_1} \bullet \mathbf{E}[(\lambda \mathbf{x}.\mathbf{t_1})] : \boldsymbol{\sigma}_1 = \mathbf{S_1}'$$

    and that:

$$\mathbf{S_2} = \mathbf{N_2}[\mathsf{n_i} \mapsto (\lambda \mathbf{x}.\mathbf{t_2})] \Vdash \mathbf{\Sigma_1}, \mathbf{E} : \boldsymbol{\sigma}_1 \to \boldsymbol{\sigma}_2 \overset{@\mathsf{n_i}}{\Longrightarrow} \mathbf{N_2} \Vdash \mathbf{\Sigma_2} \bullet \mathbf{E}[(\lambda \mathbf{x}.\mathbf{t_2})] : \boldsymbol{\sigma}_1 = \mathbf{S_2}'$$

    From the definition of $\approx_N$ it follows that $(\lambda \mathbf{x}.\mathbf{t_1}) \approx_{\mathbf{s}} (\lambda \mathbf{x}.\mathbf{t_2})$. By the definition of $\approx_{\mathbf{\Sigma}}$ we have that $\mathbf{E}[(\lambda \mathbf{x}.\mathbf{t_2})] \approx_s \mathbf{E}[(\lambda \mathbf{x}.\mathbf{t_1})]$. We conclude that the thesis holds.
  - $\gamma^+ = \gg (\lambda \mathsf{x.t})\ \mathsf{v}$: By the LTS we have that:

$$\mathbf{S_1} = \mathbf{N_1} \Vdash \mathbf{\Sigma_1} \bullet \mathbf{E}_1[(^{\boldsymbol{\sigma}_\mathbf{f}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t})\ \mathbf{v_1})] : \boldsymbol{\sigma} \overset{\gg (\lambda \mathsf{x.t});\ \mathsf{v}}{\Longrightarrow} \mathbf{N}_\mathbf{1}' \Vdash \mathbf{\Sigma_1}, \mathbf{E}_1 : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma} = \mathbf{S_1}'$$

    and that:

$$\mathbf{S_2} = \mathbf{N_2} \Vdash \mathbf{\Sigma_2} \bullet \mathbf{E}_2[(^{\boldsymbol{\sigma}_\mathbf{f}}\mathbf{S}\mathsf{A}\ (\lambda \mathsf{x.t})\ \mathbf{v_2})] : \boldsymbol{\sigma} \overset{\gg (\lambda \mathsf{x.t});\ \mathsf{v}}{\Longrightarrow} \mathbf{N}_\mathbf{2}' \Vdash \mathbf{\Sigma_2}, \mathbf{E}_2 : \boldsymbol{\sigma}_2 \to \boldsymbol{\sigma} = \mathbf{S_2}'$$

    By the LTS we know that $\mathbf{N}_\mathbf{1}' \neq \mathbf{N_1}$ and $\mathbf{N}_\mathbf{2}' \neq \mathbf{N_2}$ if and only if the argument v is produced by the same label $\mathsf{n_i}$. By the case $\gamma^+ = \mathsf{n_i}$ we conclude that $\mathbf{N}_\mathbf{1}' \approx_N \mathbf{N}_\mathbf{2}'$. We have that $\mathbf{E}_1 \approx_s \mathbf{E}_2$ as by $\mathbf{S_1} \mathcal{R} \mathbf{S_2}$ we know that they derive from two $\approx_s$ bisimilar terms. We conclude that the thesis holds.
- For case (3) we have that: mutatis mutandis.

A.5 Compiler Reflection

Proof Sketch. Proof sketch of Lemma 6:

$$\mathbf{t_1}^{\downarrow} \approx_{+} \mathbf{t_2}^{\downarrow} \Rightarrow \mathbf{t_1} \approx_{s} \mathbf{t_2}$$

We prove the lemma by the contrapositive, the lemma is restated as:

$$\mathbf{t_1} \not\approx_{s} \mathbf{t_2} \Rightarrow \mathbf{t_1}^{\downarrow} \not\approx_{+} \mathbf{t_2}^{\downarrow}$$

The proof proceeds by induction because if we have that $\mathbf{t_1} \not\approx_s \mathbf{t_2}$ then there is a finite sequence of LTS transitions until the bisimilarity $\approx_s$ fails to hold. There are two base cases:

1. $\mathbf{t_1} \overset{\gamma}{\Rightarrow} \mathbf{t_1}' \wedge \nexists \mathbf{t_2}'.\ \mathbf{t_2} \overset{\gamma}{\Rightarrow} \mathbf{t_2}'$: We proceed by case analysis over the label $\boldsymbol{\gamma}$:
   - $\mathbf{true}$: We have that: $\mathbf{t_1} \rightarrow^* \#\mathbf{t} \overset{\mathbf{true}}{\Longrightarrow} \mathbf{t_1'}$. By the assumption, $\mathbf{t_1} \not\approx_s \mathbf{t_2}$ and the LTS we have that: $\mathbf{t_2} \rightarrow^* \mathbf{v_2}$ where $\mathbf{v_2} \neq \#\mathbf{t}$. By Lemma 4 we have that $\mathbf{t_1}^{\downarrow} \twoheadrightarrow^* \#\mathbf{t}^{\downarrow}$ and $\mathbf{t_2}^{\downarrow} \twoheadrightarrow^* \mathbf{v_2}^{\downarrow}$. We conclude from the λ^+-calculus LTS that the thesis holds.
   - $\mathbf{false}$: analogous to `true`.
   - $@\mathbf{v}$: In this case $\mathbf{t_1} \rightarrow^* (\lambda \mathbf{x}.\mathbf{t_{11}}) \overset{@\mathbf{v}}{\Longrightarrow} \mathbf{t_1'}$ and $\mathbf{t_2} \rightarrow^* \mathbf{v_2}$ where $\mathbf{v_2} \neq (\lambda \mathbf{x}.\mathbf{t_{22}})$.
     By Lemma 4 we have that $\mathbf{t_1}^{\downarrow} \rightarrow^* (\lambda \mathbf{x}.\mathbf{t_{11}})^{\downarrow}$ and $\mathbf{t_2}^{\downarrow} \twoheadrightarrow^* \mathbf{v_2}^{\downarrow}$. We conclude from the λ^+-calculus LTS that the thesis holds.
2. $\mathbf{t_2} \overset{\gamma}{\Rightarrow} \mathbf{t_2}' \wedge \nexists \mathbf{t_1}'.\ \mathbf{t_1} \overset{\gamma}{\Rightarrow} \mathbf{t_1}'$: Similar to case 1.

By the LTS of the λ_s-calculus we have that if:

$$\mathbf{t_1} \overset{\gamma}{\Rightarrow} \mathbf{t_1}' \ \wedge\ \mathbf{t_2} \overset{\gamma}{\Rightarrow} \mathbf{t_2}' \ \wedge\ \mathbf{t_1}' \not\approx_s \mathbf{t_2}'$$

then $\boldsymbol{\gamma} = @\mathbf{v}$, because if $\boldsymbol{\gamma} = \mathbf{true}$ or $\mathbf{false}$ then $\mathbf{t_1}' \approx_s \mathbf{t_2}'$. We thus have one inductive case. Given:

$$\mathbf{t_1} \overset{\overline{@\mathbf{v}}, @\mathbf{v}}{\Longrightarrow} \mathbf{t_1^{n+1}} \ \wedge\ \mathbf{t_2} \overset{\overline{@\mathbf{v}}, @\mathbf{v}}{\Longrightarrow} \mathbf{t_2^{n+1}} \ \wedge\ \mathbf{t_1^{n+1}} \not\approx_s \mathbf{t_2^{n+1}}$$

We must show that

$$\mathbf{t_1}^{\downarrow} \overset{\overline{\gamma^+}, \gamma^+}{\Longrightarrow} \mathbf{S}_1^{n+1} \ \wedge \mathbf{t_2}^{\downarrow} \overset{\overline{\gamma^+}, \gamma^+}{\Longrightarrow} \mathbf{S}_2^{n+1} \ \wedge\ \mathbf{S}_1^{n+1} \not\approx_{+} \mathbf{S}_2^{n+1}$$

By the inductive hypothesis:

$$\mathbf{t_1} \overset{\overline{@\mathbf{v}}}{\Longrightarrow} \mathbf{t_1^n} \ \wedge\ \mathbf{t_2} \overset{\overline{@\mathbf{v}}}{\Longrightarrow} \mathbf{t_2^n} \ \wedge\ \mathbf{t_1^n} \not\approx_s \mathbf{t_2^n} \ \Rightarrow$$
$$\mathbf{t_1}^{\downarrow} \overset{\overline{\gamma^+}}{\Longrightarrow} \mathbf{S}_1^n \ \wedge \mathbf{t_2}^{\downarrow} \overset{\overline{\gamma^+}}{\Longrightarrow} \mathbf{S}_2^n \ \wedge\ \mathbf{S}_1^n \not\approx_{+} \mathbf{S}_2^n$$

By the λ_s-calculus LTS rule Label-App, the λ^+-calculus LTS rules O-Name and Call-N and the observation that for any value $\mathbf{v}$ there is an equivalent value v we have that for any for any $(\lambda \mathbf{x}.\mathbf{t})$:

$$(\lambda \mathbf{x}.\mathbf{t}) \overset{@\mathbf{v}}{\Longrightarrow} ((\lambda \mathbf{x}.\mathbf{t})\ \mathbf{v}) \iff$$
$$\mathbf{N} \Vdash (\lambda \mathbf{x}.\mathbf{t}) \overset{\mathsf{n_i}}{\Rightarrow} \overset{\gg \mathsf{n_i}\ ;\ \mathsf{v}}{\Longrightarrow} \mathbf{N}, [\mathsf{n_i} \mapsto (\lambda \mathbf{x}.\mathbf{t})] \Vdash ((\lambda \mathbf{x}.\mathbf{t})\ \mathbf{v})$$

This in conjunction with the IH implies the thesis.